# CS390S, Week 9:Meta-Character Vulnerabilities

Pascal Meunier, Ph.D., M.Sc., CISSP
February 20,  2008
Developed thanks to support and contributions from Symantec Corporation, support from the NSF SFS Capacity Building Program (Award Number 0113725) and the Purdue e-Enterprise Center

## What Are Meta-Characters?

- Encountered in mixed code and data channels
- They have meaning beyond that of the character they represent
  - Delimiters between data fields or between data and commands
  - In code: Syntax significant characters
    - e.g., separate or terminate commands
    - wildcards
    - operators (UNIX pipe "|", etc...)
    - etc...
- "Special elements" (c.f. CWE) is used to denote character combinations (tagwords) that have meta-character functionality

## Meta-Characters in the CWE

- Special elements (characters or reserved words), CWE ID 138
  - Quoting elements " ' (149)
  - Control characters and escape sequences (150)
  - Delimiters (tabs, commas, etc...) (140)
  - Input terminators (147)
  - Wildcards (155)
  - Dots (no CWE ID)
  - <, >, |, ;, /
  - Comment element (151)
  - etc...

PURDUE UNIVERSITY

## Mixed Data and Code Examples

- Wrappers to system calls
  - Command vs arguments
  - subshells, command substitution ("`")
  - other shell metacharacters
- HTML vs JavaScript
  - "`<script>`"
  - "`on` *`eventname`*"
- Format Strings
  - Special format specifiers
- SQL (Simple Query Language for databases)

## Meta-character vulnerabilities enable:

- Code Injection attacks, CWE ID 77
  - Shell commands, CWE ID 77
    - Related to the wrapper problem
  - SQL, CWE ID 89
  - XPATH (no CWE ID yet)
  - Custom special character injection, CWE ID 92
  - Escape, meta, or control character/sequence, CWE ID 150
  - LDAP, CWE ID 90
  - Direct dynamic code evaluation, CWE ID 95

## Meta-character vulnerabilities enable: (cont.)

- Data attacks:  add, remove, modify data
  - XML, CWE ID 91
  - HTML (no CWE ID)
- Most cross-site scripting attacks (XSS) CWE ID 79
- Related attacks
  - Format string vulnerabilities (due to '%')
  - Directory traversal, a.k.a. "dot-dot" attacks

## Related Concepts

- Character encoding and decoding
  - Character sets
- Input Validation
- Preventative techniques
  - Character escaping
  - Prepared statements and stored procedures

## Motivation: Understanding Code Injection

- Goal: trick program into executing an attacker's code by clever input construction that mixes code and data
- Mixed code and data channels have special characters that trigger a context change between data and code interpretation
  - The attacker wants to inject these meta-characters through some clever encoding or manipulation, so supplied data is interpreted as code

## Basic Example by Command Separation

- cat > example
  - `#!/bin/sh`
  - `A=$1`
  - `eval "ls $A"`
- Permissions for file "confidential" before exploit:

  -
    ```
    % ls -l confidential
    -rwxr-x---  1 pmeunier  pmeunier
    confidential
    ```
- Allow execution of "example":
  - `% chmod a+rx example`
- Exploit (what happens?)
  - `%./example ".;chmod o+r *"`

## Results

- Inside the program, the eval statement becomes equivalent to:
- `eval "ls .;chmod o+r *"`
- Permissions for file "confidential" after exploit:

  -
    ```
    % ls -l confidential
    -rwxr-xr--  1 pmeunier  pmeunier
    confidential
    ```
- Any statement after the ";" would also get executed, because ";" is a command separator.
- The data argument for "ls" has become code!

## A Vulnerable Program

-
  ```
  int main(int argc, char *argv[], char **envp)
  {
     char buf [100];
     buf[0] = '\0';
     snprintf(buf, sizeof(buf), "grep %s
  text",argv[1]);
     system(buf);
     exit(0);
  }
  ```
  What happens when this is run?
  % ./a.out \`./script\`

## Answer

- The program calls
  - `system (“grep `./script` text”);`
  - You can verify by adding "`printf( "%s", buf)`" to the program
- So we could make a.out execute any program we want
  - Imagine that we provide the argument remotely
  - What if a.out runs with root privileges?

## SQL Injection

- SQL uses single and double quotes to switch between data and code.
- Semi-colons separate SQL statements
- Example query:

  -
    ```
    "UPDATE users
    SET prefcolor='red'
    WHERE uid='joe';"
    ```
- This command could be sent from a web front-end to a database engine.
- The database engine then interprets the command

## Dynamic SQL Generation

- Web applications typically dynamically generate the necessary database commands by manipulating strings
- Example query generation:

  -
    ```
    $q = "UPDATE users
    SET prefcolor='$INPUT[color]'
    WHERE uid='$auth_user'";
    ```
- Where the value of "`$INPUT[color]`" would be originating from the client web browser, through the web server.
- And where the value for "`$auth_user`" would have been stored on the server and verified through some authentication scheme

## Client Web Browser

- Forms in client browsers return values to the web server through either the POST or GET methods
  - "GET" results in a url with a "?" before the values of the form variables are specified:
    - http://www.example.com/script?color=red
    - The value of "`$INPUT[color]`" is set to "red" in the script
- "GET" urls are convenient to hack, but there isn't any significant difference in the security of either "GET" or "POST" methods because the data comes from the client web browser regardless and is under the control of the remote attacker

## The SQL Table

- Tables are used to store information in fields (columns) in relation to a key (e.g., "uid")
- What other fields could be of interest?

-
  ```
  CREATE TABLE users (
    prefcolor varchar(20),
    uid VARCHAR(20) NOT NULL,
    privilege ENUM('normal',
  'administrator'),
    PRIMARY KEY (uid)
  );
  ```

## A Malicious SQL Query

- What if we could make the web server generate a query like:

  -
    ```
    "UPDATE users
    SET prefcolor='red',
    privilege='administrator'
    WHERE uid='joe';"
    ```
- Can we engineer the value of "color" given to the web server so it generates this query?
  - Note how code and data are mixed in the same channel
    - ❖ Better database interfaces provide separate channels
      - Java prepared statements
      - Stored procedures

## Malicious HTTP Request

- `http://www.example.com/script?color=red',privilege='administrator`
- The "color" input is then substituted to generate SQL:

  -
    ```
    $q = "UPDATE users
    SET prefcolor='$INPUT[color]'
    WHERE uid='$auth_user'";
    ```
- It gives the query we wanted!

## Results

- Joe now has administrator privileges.

## Adding Another SQL Query

- Let's say Joe wants to run a completely different query:
  - "DELETE FROM users"
    - This will delete all entries in the table!
- How can the value of "color" be engineered?

## Malicious HTTP Request

- `http://www.example.com/script?color=red'%3Bdelete+from+users%3B`
  - %3B is the url encoding for ";"
- What happens when the "color" input is used to generate SQL?

  -
    ```
    $q = "UPDATE users
    SET prefcolor='$INPUT[color]'
    WHERE uid='$auth_user'";
    ```

## Result

-
  ```
  UPDATE users
  SET prefcolor='red';
  delete from users;
  WHERE uid='$auth_user'";
  ```
- The last line generates an error, but it's already too late;  all entries have been deleted.
- The middle query could have been anything

## FAQs

- Couldn't the database have a separate account for "Joe" with only the privileges he needs (e.g., no delete privilege)?
  - In theory yes, but in practice the management of such accounts and privileges, and connecting to the database with the correct IDs, adds significant complexity
    - Most often a database account is created for the entire web application, with appropriate limitations (e.g., without privileges to create and drop tables)
    - A good compromise is to create database accounts for each class of user or class of operation, so:
      - if Joe is a regular user he wouldn't have delete privileges for the user table
      - Changing user preferences, as an operation type, doesn't require delete privileges

## FAQs

- Doesn't SSL protect against this sort of attack?
  - No
- But what if you authenticate users with a username/password over SSL? Then, if the user does SQL injection, the server admins will know who perpetrated the crime, right?
  - Not necessarily; only if you have sufficient audit logging.

## Other SQL Injection Methods

- Let's say you've blocked single quotes, double quotes and semi-colons.
- What else can go wrong?
  - How about "\"?
  - If attacker can inject backslashes, then escaped quotes could get ignored by the database

## PHP-Nuke SQL injection CVE-2002-1242

- iDefense advisory dated Oct. 31, 2002
- Malicious url:
  - modules.php?
    name=Your_Account&op=saveuser&uid=2&bio=
    %5c&EditedMessage=
    no&pass=xxxxx&vpass=xxxxx
    &newsletter=,+pass=md5(1)/*
- %5c is the encoding for '\'

## Let's Look at the SQL

```
UPDATE nuke_users
   SET name        = '', email       = '',
       femail      = '', url         = 'http://',
       pass        = 'xxxxx', bio           = '\',
       user_avatar = '',
       user_icq    = '',
       user_msnm   = '',
       newsletter  = ',
       pass=md5(1)/*' WHERE uid='2'
```

- Notice how bio would be set according to the text in red?
  - '' (two single quotes) make the database insert a single quote in the field, effectively the same as \'
- Notice how the comment field, '/*', is used to comment out the "WHERE" clause for the uid? This means that the query applies to **all** users!

## What Happened?

- All passwords were changed to the value returned by the function "md5(1)"
  - Constant: "c4ca4238a0b923820dcc509a6f75849b"
- Attacker can now login as anyone

## The Input Cleansing Idea

- Model the expected input
  - Discard what doesn't fit (e.g., metacharacters)
- Intuitive Approach
  - Delete metacharacters from the input
    - Need a complete list
  - Problems:
    - Character encodings
      - octal, hexadecimal, UTF-8, UTF-16...
    - Obfuscation
    - Escaped characters that can get interpreted later
    - Engineered strings such that by blocking a character, something else is generated

## CWE: Consequences of Bad Input Cleansing

- Collapse of Data into Unsafe Value (182)
- Examples:
  - CVE-2005-3123 - "/.//..//////././" is collapsed into "/.././" after ".." and "//" sequences are removed.
  - CAN-2005-2169 - Regular expression intended to protect against directory traversal reduces ".../...//" to "../"
- Path Issue - doubled dot dot slash - '....//' (34)
  - Cleansing error that removes a single "../" from "....//"
- Path Issue - doubled triple dot slash - '.../...//' (35)
  - CAN-2005-0202 - ".../....///" bypasses regexp's that remove "./" and "../"

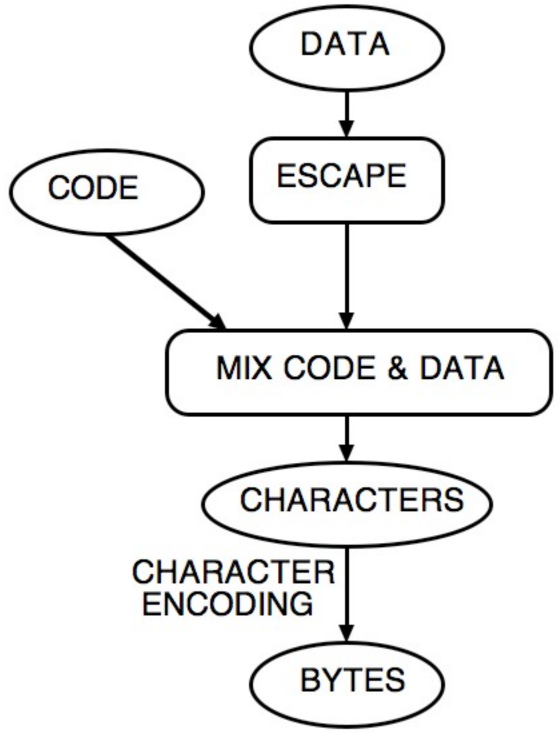
## Input Cleansing and Sanitization

- Error prone
- Complex
- May be too crude (loss of functionality)
- Black List approach
- Instead of trying to pick valid parts of the input and to recover from attacks in the input, it is safer to simply reject input identified as incorrect (and potentially malicious)

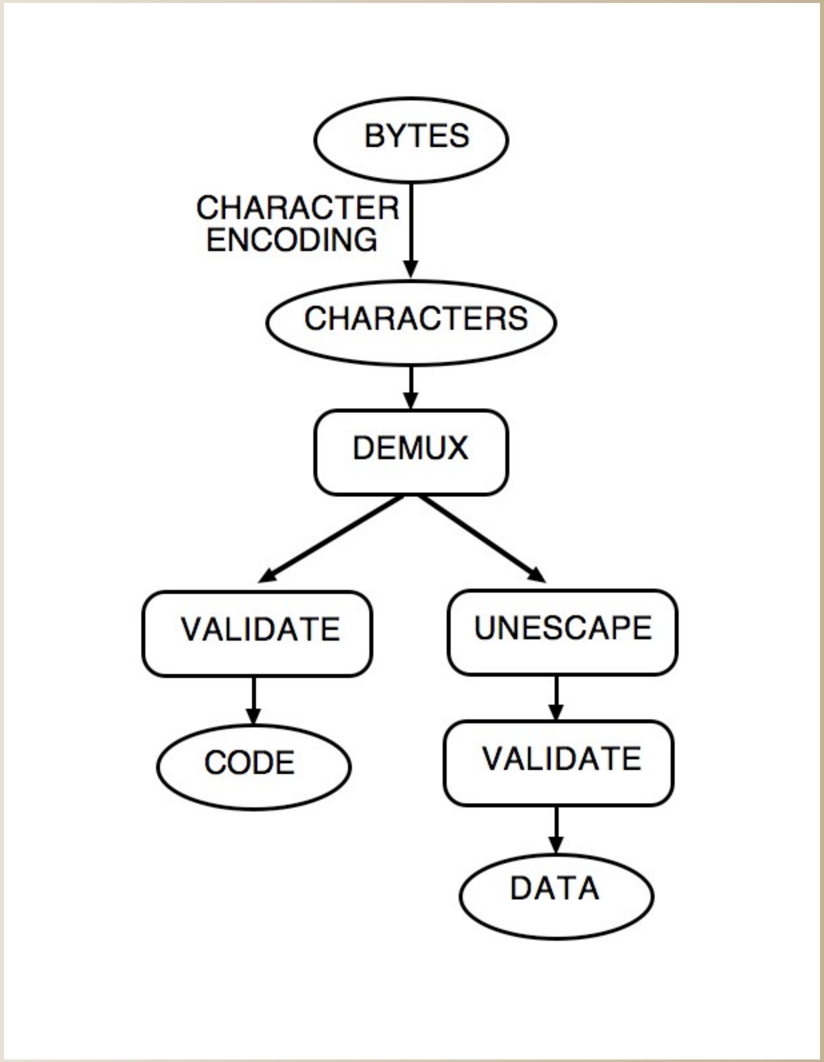
## Problem Definition

- Need to transmit data
- Need to transmit commands, or data field names, separators, tags, etc...
- Only one channel!
- Meta-characters can separate data from commands and have syntactic significance
  - What to do if the data itself can contain meta-characters?
    - "Escape"!
      - Adopt an equivalent representation without the meta-character meaning
- Characters can be wide or multi-byte

## Sending Data



## Receiving Data



## Escaping Characters

- Definition: removal of the meta-character function and meaning, so that a character can be used in the data
- Example:  slash escapes
  - \' is an escaped single quote
- Example: HTML escape
  - &#39; is an escaped single quote
  - &quot; represents an escaped double quote (")

## URL-encoding, CWE ID 177

- a.k.a Hex encoding
- %XX where XX is an hexadecimal number
  - %27 is an escaped single quote
- Misnomer: this is not an encoding, but an escape mechanism
- "URL-encoding" can be used with *any* character encoding
  - The character is then escaped by replacing each byte with a %XX value
  - Given a sequence of URL-encoded bytes, there is no way to know which character encoding was used...
    - Each part of a URL may use a different character encoding

## Escaping Multi-Byte Characters

- Using slashes to escape multi-byte (e.g., unicode, CWE ID 176) characters is dangerous
- Attack Scenario:
  - A byte indicates a character that is two bytes long
  - Second byte is %27 (single quote)
  - Sender program escapes quote by adding a slash
  - Receiver program sees a two-byte character followed by a quote
    - SQL injection!
- PostGreSQL now rejects invalid multi-byte encodings ending with a backslash (c.f. CVE-2006-2314) in an attempt to block exploits

## Other Issues

- Decoding using a different character encoding than the one used for encoding
  - Alternate Encoding, CWE ID 173
  - Meta-characters could slip through this way
- Multiple unescape mechanisms
  - `"%2527"` decodes to "' " (a single quote) after two decoding steps; this allowed the execution of arbitrary SQL commands in Phorum 3.4.7 (CVE-2004-1938)
  - Fooling of the security zone mechanism in Internet Explorer (CVE-2005-0054) with doubly-encoded domain names
  - Double Encoding, CWE ID 174

## Questions or Comments?