NECESSARY BACKGROUND

ON

MEMORY EXPLOITS AND

WEB APPLICATION VULNERABILITIES

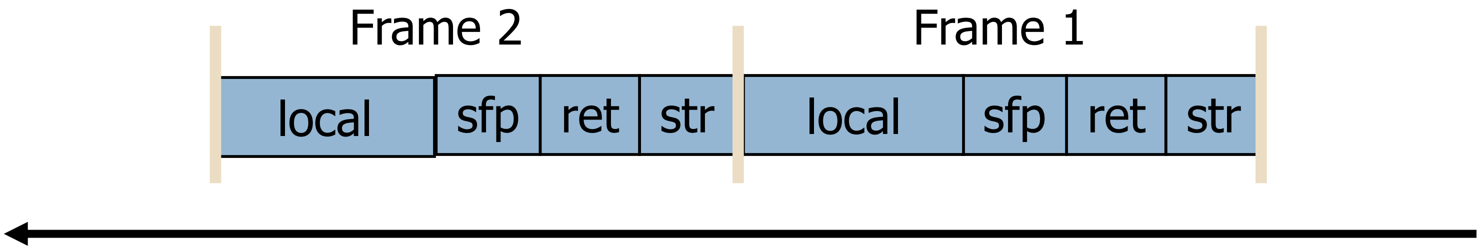
Outline

- Memory safety attacks
  - Buffer overruns
  - Format string vulnerabilities

- Web application vulnerabilities
  - SQL injections
  - Cross-site scripting attacks

Buffer Overflows

Buffer Overrun Example



```
void lame (void) {
        char small[30];
        gets(small);
        printf("%s\n", small);
}
```

Input Validation

- Classifying vulnerabilities:
  - Buffer overflows can be viewed as an example of *improper input validation*
  - Another related type of vulnerability is *information leaks*

- Other notable examples:
  - Format string vulnerabilities
  - SQL injection attacks
  - Cross-site scripting attacks

- Mechanisms to prevent attacks
  - Better input validation
  - Safe programming techniques
  - Techniques for detecting potential buffer overflows in code
    - Static analysis
    - Runtime analysis
    - Fuzzing/penetration testing
    - Write-box fuzzing
    - etc.

Secure Programming Techniques

- Validate all input
  - Easier said than done
  - Why is that?

- Avoid buffer overflows
  - Use safe string manipulation functions
  - Careful length checking
  - Avoid statically declared arrays
  - etc.

- Or use a memory-safe language
  - Java or C#
  - JavaScript (not type-safe)

Validating Input

- Determine acceptable input, check for match --- don’t just check against list of “non-matches”
  - Limit maximum length
  - Watch out for special characters, escape chars.

- Check bounds on integer values
  - Check for negative inputs
  - Check for large inputs that might cause overflow!

Avoid strcpy, …

- We have seen that **strcpy** is unsafe
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until “\0” is encountered, ignoring the size of buf

- Avoid **strcpy**(), **strcat**(), **gets**(), etc.
  - Use **strncpy**(), **strncat**(), instead
  - Still, computing proper bounds is difficult in practice
  - Easy to mess up, off-by-one errors are common

Static and Dynamic Analysis

- Static analysis: run on the source code prior to deployment; check for known flaws
  - e.g., flawfinder, cqual
  - Or Prefix/Prefast
  - Or Coverity or Fortify tools
  - Will look at some more recent work in this course as well as older stuff

- Dynamic analysis: try to catch (potential) buffer overflows during program execution
  - Soundness
  - Precision

- Comparison?
  - Static analysis very useful, but not perfect
    - False positives
    - False negatives
  - Dynamic analysis can be better (in tandem with static analysis), but can slow down execution
  - Historically of great importance, drove adoption of type-safe languages such as Java and C#

Dynamic analysis: Libsafe

- Very simple example of what can be done at runtime

- Intercepts all calls to, e.g., **strcpy**(dest, src)
  - Validates sufficient space in current stack frame:
    |frame-pointer – dest| > strlen(src)
  - If so, executes **strcpy**; otherwise, terminates application

Preventing Buffer Overflows

- Operating system support:
  - Can mark stack segment as non-executable
  - Randomize stack location

- Problems:
  - Does not defend against `return-to-libc’ exploit
    - Overflow sets ret-addr to address of libc function
  - Does not prevent general buffer overflow flaws, or heap overflow

- Basic heap overflows can be helped with ALSR

Heap-based Buffer Overruns and Heap Spraying

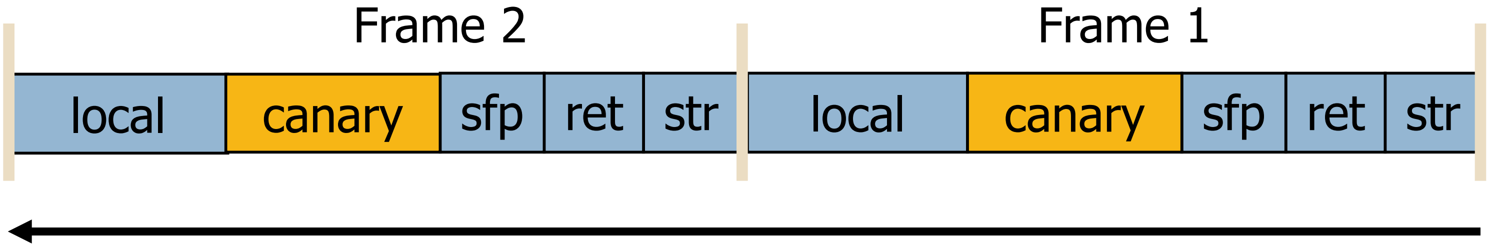
- Buffer overruns consist of two steps
  - Introduce the payload
  - Cause the program to jump to it

- Can put the payload/shellcode in the heap
  - Arbitrary amounts of code
  - Doesn’t work with heap randomization
  - Location of the payload changes every time

- Heap spraying:
  - Allocate multiple copies of the payload
  - When the jump happens, it hits the payload with a high probability

StackGuard

- Embed random “canaries” in stack frames and verify their integrity prior to function return
- This is actually used!
- Helpful, but not foolproof…



More Methods …

- Address obfuscation
  - Encrypt return address on stack by XORing with random string. Decrypt just before returning from function
  - Attacker needs decryption key to set return address to desired value

More Input Validation Flaws

Format String Vulnerabilities

- What is the difference between

  `printf(buf);`

  and

  `printf(“%s”, buf);`

  ?
- What if buf holds %x ?
- Look at memory, and what printf expects…

Format String Exploits

- Technique:
  - Declare a variable of type **int** in line 4 and call it bytes_formatted
  - Line 6 the **format string** specifies that 20 characters should be formatted in hexadecimal (“%.20x”) using buffer
  - When this is done, due to the “%n” specifier write the value 20 to bytes_formatted
- Result:
  - This means that we have written a value to another memory location
  - Very definition of violating memory safety
  - May be possible to gain control over a program’s execution

```
#include <stdio.h>
int main() {
    int bytes_formatted=0;
    char
buffer[28]=”ABCDEFGHIJKLMNOPQRSTUVWXYZ”;

    printf(“%.20x%n”,buffer,&bytes_formatted);
    printf(
     “\nThe number of bytes formatted in the
previous printf statement
                    was %d\n”,bytes_formatted);
   return 0;
}
```

Other Input Validation Bugs

- Integer overflow…
- Consider the code:

```
strncpy(msg+offset, str, slen);
```

where the adversary may control `offset`

- By setting the value high enough, it will wrap around and be treated as a negative integer!
- Write into the msg buffer instead of after it

Web Application Vulnerabilities

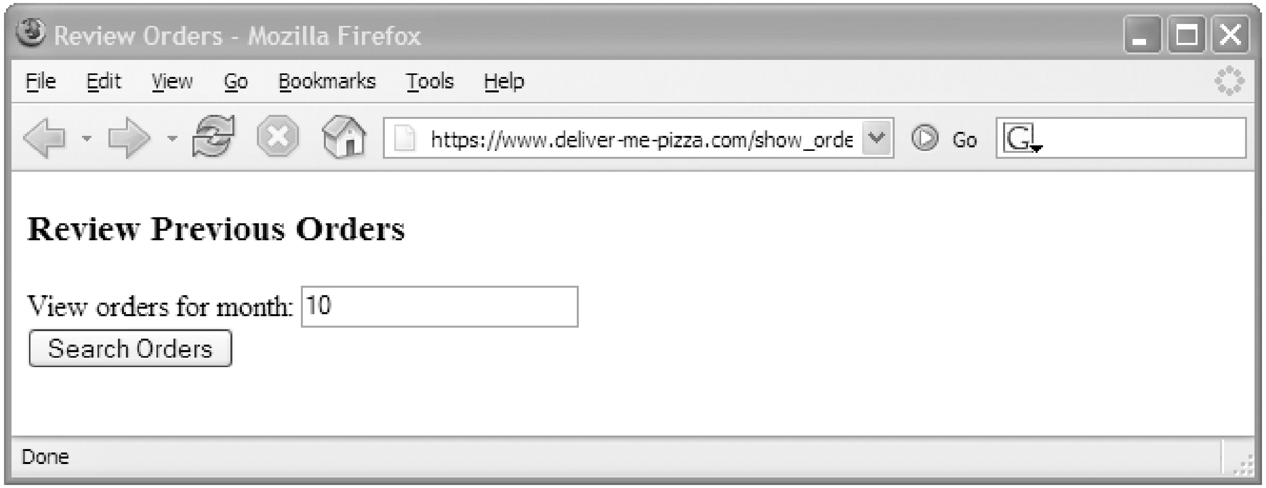
SQL Injection Attacks

- Affect applications that use untrusted input as part of an SQL query to a back-end database

- Specific case of a more general problem: using untrusted input in commands

SQL Injection: Example

- Consider a browser form, e.g.:



Review Previous Orders

View orders for month: 10

Search Orders

- When the user enters a number and clicks the button, this generates an http request like
  https://www.pizza.com/show_orders?month=10

Example Continued…

- Upon receiving the request, a Java program might produce an SQL query as follows:

```
sql_query
      = "SELECT pizza, quantity, order_day "
          + "FROM orders "
          + "WHERE userid=" + session.getCurrentUserId()
          + " AND order_month= "
          + request.getParameter("month");
```

- A normal query would look like:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=10
```

Example Continued…

- What if the user makes a modified http request:
  https://www.pizza.com/show_orders?month=0%20OR%201%3D1
- (Parameters transferred in URL-encoded form, where meta-characters are encoded in ASCII)
- This has the effect of setting

  ```
  request.getParameter(“month”)
  ```

  equal to the string

  ```
  0 OR 1=1
  ```

Example Continued

- So the script generates the following SQL query:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE (userid=4123
AND order_month=0) OR 1=1
```

- Since AND takes precedence over OR, the above always evaluates to TRUE
  - The attacker gets every entry in the database!

Even Worse…

- Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 OR 1=0
UNION SELECT cardholder, number, exp_date
FROM creditcards
```

- Attacker gets the entire credit card database as well!

More Damage…

- SQL queries can encode multiple commands, separated by ‘;’
- Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 ;
DROP TABLE creditcards
```

- Credit card table deleted!
  - DoS attack

More Damage…

- Craft an http request that generates an SQL query like the following:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 ;
INSERT INTO admin VALUES (‘hacker’, ...)
```

- User (with chosen password) entered as an administrator!
  - Database owned!

May Need to be More Clever…

- Consider the following script for *text* queries:

```
sql_query
      = "SELECT pizza, quantity, order_day "
          + "FROM orders "
          + "WHERE userid=" + session.getCurrentUserId()
          + " AND topping= ‵ "
          + request.getParameter(“topping") + “’”
```

- Previous attacks will not work directly, since the commands will be quoted

- But easy to deal with this…

Example Continued…

- Craft an http request where

```
request.getParameter("topping")
```

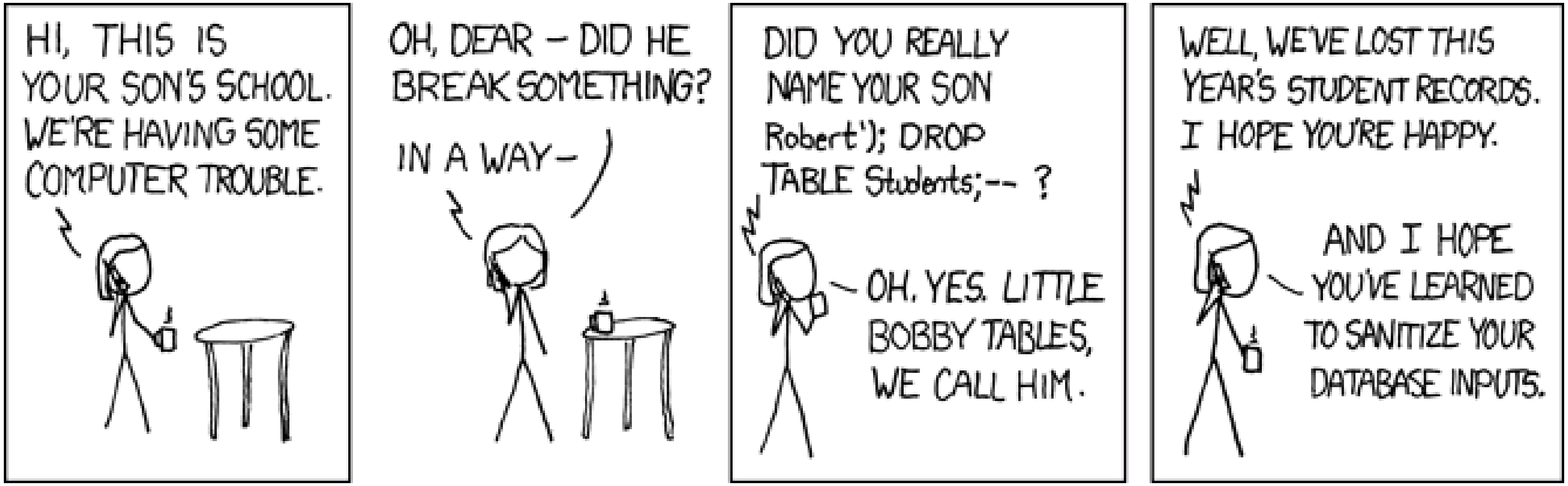
is set to

```
abc'; DROP TABLE creditcards; --
```

- The effect is to generate the SQL query:

```
SELECT pizza, quantity, order_day
FROM orders
WHERE userid=4123
AND toppings='abc';
DROP TABLE creditcards ; --'
```

- ('--' represents an SQL comment)



Source: http://xkcd.com/327/

Solutions?

- Blacklisting
- Whitelisting
- Encoding routines
- Prepared statements/bind variables
- Mitigate the impact of SQL injection

Blacklisting?

- I.e., searching for/preventing ‘bad’ inputs
- E.g., for previous example:

```
sql_query
      = "SELECT pizza, quantity, order_day "
          + "FROM orders "
          + "WHERE userid=" + session.getCurrentUserId()
          + " AND topping= ‘ "
          + kill_chars(request.getParameter(“topping"))
          + “’”
```

- …where kill_chars() deletes, e.g., quotes and semicolons

Drawbacks of Blacklisting

- How do you know if/when you've eliminated all possible 'bad' strings?
  - If you miss one, could allow successful attack
- Does not prevent first set of attacks (numeric values)
  - Although similar approach could be used, starts to get complex!
- May conflict with functionality of the database
  - E.g., user with name O'Brien

Whitelisting

- Check that user-provided input is in some set of values known to be safe
    - E.g., check that month is an integer in the right range

- If invalid input detected, better to reject it than to try to fix it
    - Fixes may introduce vulnerabilities
    - *Principle of fail-safe defaults*

Prepared Statements/bind Variables

- Prepared statements*:* static queries with *bind variables*
    - Variables not involved in query parsing

- Bind variables: placeholders guaranteed to be data in correct format

A SQL Injection Example in Java

```
PreparedStatement ps =
         db.prepareStatement(
                "SELECT pizza, quantity, order_day "
                + "FROM orders WHERE userid=?
                AND order_month=?");

ps.setInt(1, session.getCurrentUserId());
ps.setInt(2,
         Integer.parseInt(request.getParameter("month")));
ResultSet res = ps.executeQuery();
```

Bind variables

There’s Even More

- **Practical SQL Injection: Bit by Bit**

- Overall, SQL injection is easy to fix by banning certain APIs
  - Prevent queryExecute-type calls with non-constant arguments
  - Very easy to automate
  - See a tool like LAPSE that does it for Java

Cross-site Scripting

- If the application is not careful to encode its output data, an attacker can inject script into the output

```
out.writeln(“<div>”);
out.writeln(req.getParameter(“name”));
out.writeln(“</div>”);
```

- name:

```
<script>…; xhr.send(document.cookie);</script>
```

- Simplest version called *reflected* or type-1 *XSS*

Memory Exploits and Web App Vulnerabilities Compared

- **Buffer overruns**
    - Stack-based
    - Return-to-libc, etc.
    - Heap-based
    - Heap spraying attacks
    - Requires careful programming or memory-safe languages
    - Don’t always help as in the case of JavaScript-based spraying
    - Static analysis tools

- **Format string vulnerabilies**
    - Generally, better, more restrictive APIs are enough
    - Simple static tools help

- **Cross-site scripting**
    - XSS-0, -1, -2, -3
    - Requires careful programming
    - Static analysis tools

- **SQL injection**
    - Generally, better, more restrictive APIs are enough
    - Simple static tools help