Secure Software Development
A Taxonomy of Coding Errors

Objectives

- Recognize through code review simple programming mistakes.
- Define the concept of a taxonomy
- Explain the relationship between kingdoms and phyla
- Critique source code for examples of coding errors
- Explain how simple coding errors might be exploited by an adversary

What is wrong with this code?

```
#include <stdio.h>
#include <string.h>

int main() {
   char longString[] = "Cellular bananular phone";
   char shortString[16];
   strncpy(shortString, longString, 16);
   printf("The last character in shortString is: %c
    %1$x\n", shortString[15]);
   printf(shortString);
   return (0);
}
```

Stack

← Look for a NULL terminator.

- Taxonomy is the practice and science of classification.

- Roots in Greek
  - *τάξις, taxis*
    - meaning 'order', 'arrangement'
  - *νόμος, nomos*
    - 'law' or 'science'.

Kingdoms and Phyla

- Definition 1.
  - A phylum represents a specific type of coding error.

- Definition 2
  - A kingdom is a collection of phyla that share a common theme.

The Kingdoms

1. Input Validation and Representation
2. API Abuse
3. Security Features
4. Time and State
5. Errors
6. Code Quality
7. Encapsulation
*. Environment

Using something incorrectly = Bad things

Dumb mistakes

2 mistakes native to machine we are on.

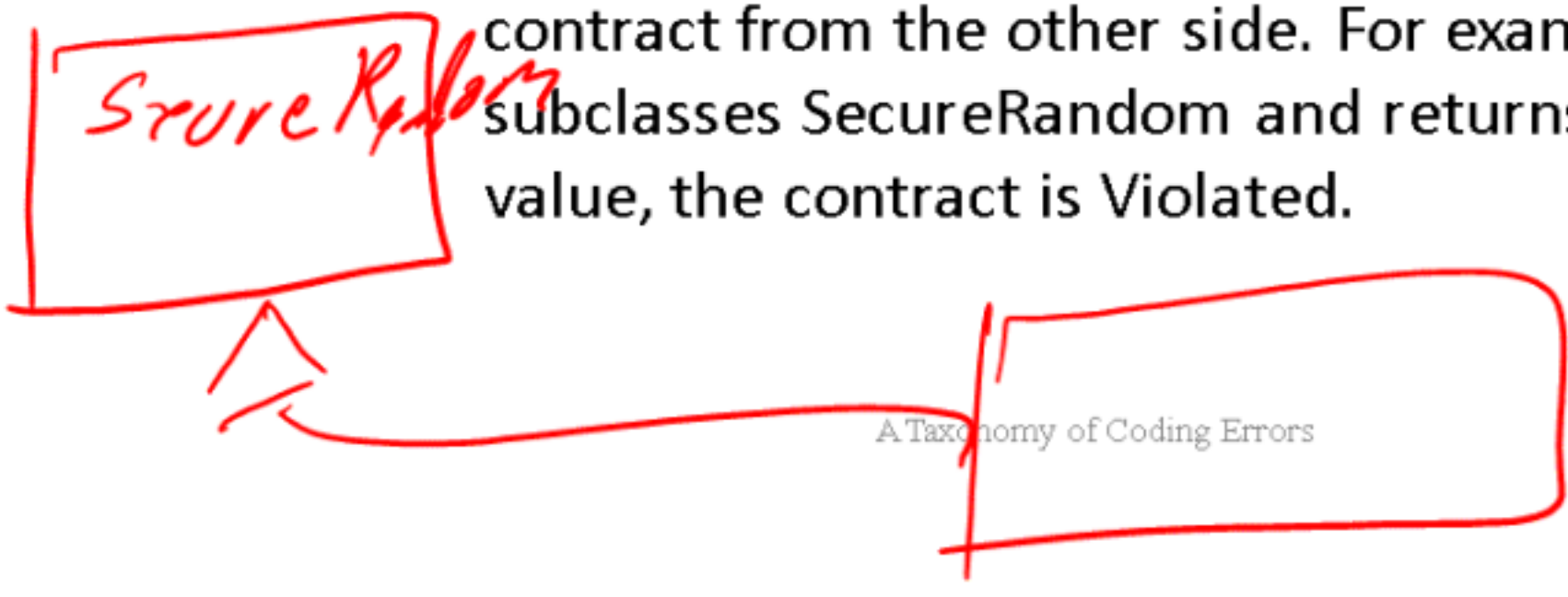
Input Validation and Representation

- Input validation and representation problems are caused by metacharacters, alternate encodings and numeric representations.
- Security problems result from trusting input. The issues include:
  - Buffer Overflows, Cross-Site Scripting attacks, SQL Injection, and many others.

Bad . . .

API Abuse

- An API is a contract between a caller and a callee. The most common forms of API abuse are caused by the caller failing to honor its end of this contract. For example, if a program fails to call chdir() after calling chroot(), it violates the contract that specifies how to change the active root directory in a secure fashion. Another good example of library abuse is expecting the callee to return trustworthy DNS information to the caller. In this case, the caller abuses the callee API by making certain assumptions about its behavior (that the return value can be used for authentication purposes). One can also violate the caller-callee contract from the other side. For example, if a coder subclasses SecureRandom and returns a non-random value, the contract is Violated.



API Abuse Example

```
#include <stdio.h>
#include <unistd.h>
#define MAXSIZE    40
Void test(char *str)
{
        char buf[MAXSIZE];
        int x;
         /* str can contain ".." components */
        snprintf(buf, sizeof buf, "/usr/games/%s", str);
        x = execl(buf, str, 0);    /* BAD */
        if(x < 0) { ; }
}
Int main(int argc, char **argv)
{
        char *userstr;
        if(argc > 1) {
                userstr = argv[1];
                test(userstr);
        }
        return 0;
}
```

API Abuse Example

```
#include <iostream>
#include <string>
using namespace std;


int main(int argc, const char *argv[])
{
        string cmd("dir ");
        if(argc>1){
            cmd.append(argv[1]);
            cout<<system(cmd.c_str())<<endl;
        }

        return 0;
}
```


Security Features

- Software security is not security software. Here we're concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management.

Using the given resources properly

Security Features

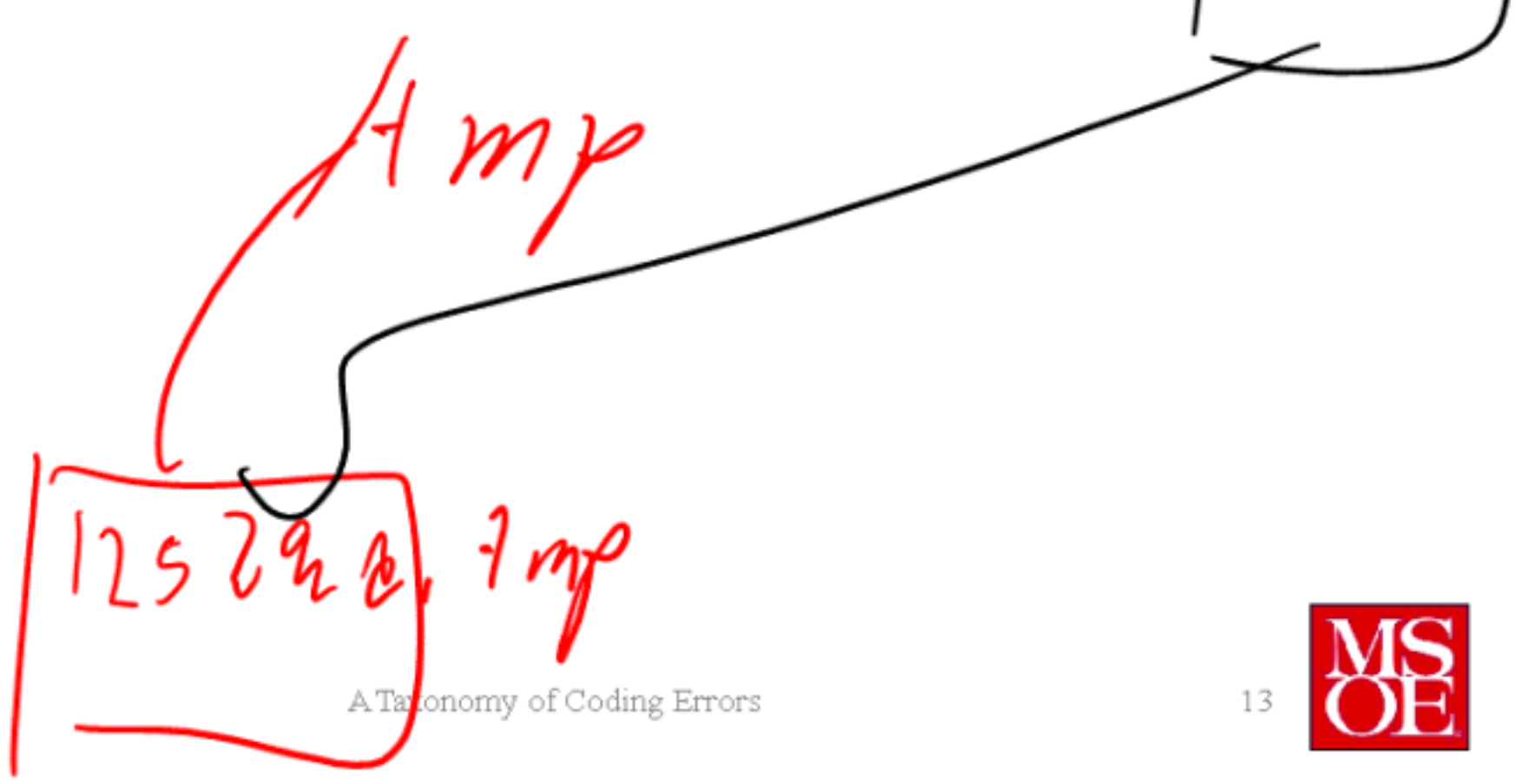
```
#include <iostream>
#include <string>
using namespace std;

int main()
{
        string userpass;
        cout << "Enter password: " << endl;
        cin >> userpass;
        if (userpass == "DEADBEEF")
          cout << "You are now identified." << endl;
        else
          cout << "Your password is not valid, please
    reenter it." << endl;
        return 0;
}
```

Time and State

- Distributed computation is about time and state. That is, in order for more than one component to communicate, state must be shared, and all that takes time.
- Most programmers anthropomorphize their work. They think about one thread of control carrying out the entire program in the same way they would if they had to do the job themselves.

Time and State : Temporary Files

- victim.c:
- filename = mktemp(template);
- fd = open(filename, ');



Error Handling

- Errors and error handling represent a class of API. Errors related to error handling are so common that they deserve a special kingdom of their own. As with API Abuse, there are two ways to introduce an error-related security vulnerability: the most common one is handling errors poorly (or not at all). The second is producing errors that either give out too much information (to possible attackers) or are difficult to handle.

Error Handling Example

```
• try {
• mysteryMethod();
• } catch (NullPointerException npe) {
• }
```

```
import java.io.*;
import java.util.*;
/** How NOT to implement a catch.  */
public final class BadCatch {
  public static void main( String... arguments ) {
    List<String> quarks = Arrays.asList(
      "up", "down", "charm", "strange", "top", "bottom"
    );
    //serialize the List
    try {
      ObjectOutputStream output =  new ObjectOutputStream(new
FileOutputStream("quarks.ser"));
    try{
      output.writeObject(quarks);
    }
    finally{
      //flush and close all streams
      output.close();
    }
  }
  catch(IOException exception){
    //TRIED OUR BEST
  }
```


Code Quality

- Poor code quality leads to unpredictable behavior. From a user's perspective that often manifests itself as poor usability. For an attacker it provides an opportunity to stress the system in unexpected ways.

Code Quality

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
void execute(short *vector)
{       for (unsigned i=0;i<3; i++)
                switch(i)
                {       case 2:
                                free(vector);
                                break;
                        default:
                             printf("%d,vector[i]);
                                break;
                }
        free(vector);
}
int main(int argc, char *argv[])
{        short *vector = (short *)NULL;
        if (!(vector = (short *)calloc(3,sizeof(short))))
        {       printf ("Allocation error!\n");
                return 0;
        }
        execute(vector);
        printf ("\n");
        return 0;
}
```


Encapsulation

- Encapsulation is about drawing strong boundaries. In a web browser that might mean ensuring that your mobile code cannot be abused by other mobile code. On the server it might mean differentiation between validated data and unvalidated data, between one user's data and another's, or between data users are allowed to see and data that they are not.

Encapsulation

```
static bool debug = false;
// Debug entry points here
void promote_root() {
  if (debug) {    // set root rights
                  printf ("# You are root now...\n");
}  }
int main(int argc, char *argv[])
{ if (argc > 1)
  { const unsigned nbArgs = argc;
    for (unsigned i=1;i<nbArgs;++i)
    { if (!strncmp(argv[i],"-debug",6))
      {
        debug = true;
        printf("Move to debugging mode\n");
// for debugging code and process, let's say you need root rights
        if (strlen(argv[i]) >= 11 && !strncmp(argv[i]+6,":root",5))
        { promote_root();
        }
  } } }
  else
  { printf("No args...\n");
  }
}
```