CSCI 455: Principles of Computer Security

Set 05:
Secure Programming, Part I.

Slide Sources

- Multiple sources including
  - Stallings & Brown textbook materials
    - Chapter 10
  - Other readings including NIST publications
  - CSCI 645 Secure Coding Slides
    - Several slides from Professor Howles
  - Others

A Quick Quiz

- Most of the top vulnerabilities are "cutting edge" problems. T/F
- C is a secure language because it provides flexibility in the language, allowing programmes to enhance security. T/F
- Java is a completely safe language. T/F
- When producing "secure" code, security must be "tested in" at the end of the life cycle. T/F

Some Secure Coding Basics

- What does “Secure Coding” mean to you?
- What is meant by “type safe” languages?
- Why is a “portable” language more likely to present security issues?
- What can a compiler do to help the programmer?
- What can the operating system do help the programmer?

Recurring Themes

- Code weak in quality (containing defects) is likely to introduce security flaws and vulnerabilities that would otherwise not be an issue
  - We can't really talk about secure code without talking about quality code
- Not all of what we'll look at are actual security issues – we'll also look at code defects and just plain bad coding habits
  - At the end of the day, does the difference between secure and quality code matter?

Basic Terminology (Recap)

- Security Policy
  - Set of rules/practices that specify or regulate how a system or organization provides security systems
- Security Mechanism
  - An implementation in code that conforms to the security policy
- Security Flaw
  - A software defect that poses a potential security risk
- Vulnerability
  - Set of conditions that allow attacker to violate an explicit or implicit security policy
- Exploit
  - Software or technique that takes advantage of a security vulnerability to violate an explicit or implicit security policy
- Mitigation
  - Methods, techniques, processes, tools or runtime libraries that can prevent or limit exploits against vulnerabilities

Then … and … Now

- Smaller systems: less code
- Computers behind closed doors
- Limited access – no networks
- Computing "experts" – few (if any) novice users
- G3 were the norm
  - Guns, gates, guards
  - Are these mechanisms still realistic?

- Personal computing – "everyone" has a computer
- Mobile, networked world
  - More and more devices
- Very large systems
- Legacy code
- Code reuse
  - Code being used in ways not originally intended
- Sacrifices for performance issues
- Regular software updates
- Crashes or outages are devastating

Securing Computers

- Deleting or destructing files
- Crashing systems
- Denying access
- Stealing information, resources or services

What does this mean for code?

- Code behaves as intended
- If attacked or compromised, the software system can resume to a safe state
- After failures, the software can resume to a safe state
- The code is resistant to failures, even if not used as designed and intended
  - This is a fundamental problem.
- This is tricky because code defects (a lack of quality) may present security flaws

Trusted Computing Base

- Parts of a system that must work correctly and reliably to ensure proper function of the system
  - e.g., OS kernel and hardware
- Principle of Least privilege – need to know
- Need to rely on languages, compilers, and OS to help enforce protection and security mechanisms

Resources You Need to Review

- SWE/SANS Top 25
  - http://www.sans.org/top25-software-errors/
- Open Web Application Security Project (OWASP)
  - https://www.owasp.org/
- DHS Build Security In
  - https://buildsecurityin.us-cert.gov/
- Resources at Mitre.org
- CMU SEI/ CERT
  - http://www.cert.org/

C Strings and Numbers

Ponder Over This

Alan Paller, director of research at the SANS Institute, expressed frustration that "everything on the [SANS Institute Top 20 Internet Security] vulnerability list is a result of poor coding, testing and sloppy software engineering. These are not 'bleeding edge' problems, as an innocent bystander might easily assume. Technical solutions for all of them exist, but they are simply not implemented."

Houston. We Have A Problem! (Is it with C and C++ ? No?)

- The Spirit of C (C Charter, point 6)
  - Trust the programmer
  - Don't prevent programmer from doing what needs to be done
  - Keep the language small and simple
  - Provide only one way to do an operation
  - Make it fast, even if portability not guaranteed
- Issues
  - Which of these impact Secure Coding?

Type Safety

"The C programming language lacks type safety"

- What is typing?
  - Typeless v. typed
  - Weakly v. strongly typed
  - Statically v. runtime typed
- Issue: explicit cast in C
- Type safety
  - Preservation
    - if a variable x has type t, and x evaluates to a value v, then v also has type t
  - Progress
    - Evaluation of an expression does not get stuck in any unexpected way: we have a value (and are done), or there is a way to proceed
  - In short, any operation on a particular type results in another value of that type
- Lack of type safety in C
  - Leads to several security flaws and vulnerabilities
- Onus
  - C programmer must develop code free from undefined behaviors, with or without compiler help

Major Mechanism for Information Interchange

- Strings are used to exchange information
  - Between users and software
  - Between software and software
- Strings usage
  - Text fields in GUIs and web applications
  - Command line arguments
  - Environment variables
- Increasing XML usage means greater impact
- Even though strings are so basic, they are not a built-in type in C or C++
  - The standard C library supports strings of type char and wide strings of type wchar_t
- Software vulnerabilities due to weaknesses in
  - String representation, management, and manipulation

Strings in C and C++

- String representation
  - 8-bit ASCII
  - Wide characters
  - Unicode characters
- Issues
  - Null termination or else
  - C versus C++ issues and mismatches
    - std::basic_string, std::string, and std::wstring
  - Not possible to avoid multiple string types in C++
    - Causes several problems for secure coding

Common String Manipulation Errors

- Improperly Bounded String Copies
- Copying and Concatenating Strings
- Off-by-One Errors
- Null termination errors
- Incorrect string truncation
- Others?

Common String Manipulation Errors

- Copying unbounded strings

  char password[80];

  gets (password);

  char buf[12]; cin >> buf;
- Solutions
  - C: malloc as needed
  - C++: set cin input field length as needed
- Off by one errors

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

int main(void) {
	char s1[] = "012345678";
	char s2[] = "0123456789";
	char *dest;
	int i;

	strcpy_s(s1, sizeof(s2), s2);
	dest = (char *)malloc(strlen(s1));
	for (i=1; i <= 11; i++) {
		dest[i] = s1[i];
	}
	dest[i] = '\0';
	printf("dest = %s", dest);
	/* ... */;
}
```

Other Common String Manipulation Errors

- String truncation
  - Insufficient space
- Null-termination errors due to lack of space

```
int main(int argc, char* argv[]) {
  char a[16];
  char b[16];
  char c[32];
  strcpy(a, "0123456789abcdef");
  strcpy(b, "0123456789abcdef");
  strcpy(c, a);
  strcat(c, b);
  printf("a = %s\n", a);
  return 0;
}
```

- Ban certain functions?
  - strcpy(), strcat(), gets(), streadd(), strecpy(), and strtrns()
- Errors without functions

```
int main(int argc, char *argv[]) {
  int i = 0;
  char buff[128];
  char *arg1 = argv[1];
  while (arg1[i] != '\0' ) {
    buff[i] = arg1[i];
    i++;
  }
  buff[i] = '\0';
  printf("buff = %s\n", buff);
}
```

Code and Software Quality

Quality?

- Code weak in quality (contains defects) is likely to introduce security flaws and vulnerabilities that would otherwise not be an issue
- When you write code, regardless of the language, what are your thoughts about:
  - Globals?
  - Module size?
  - Adhering to a coding standard?
  - Naming conventions?
  - Comments?

Foster a Quality Culture

- Think about prevention, not detection
- Cannot afford fire fighting
  - Quality issues
    - Excessive costs
    - Time
    - Rework
    - Loss of customer base; customer confidence
  - Security issues
    - You may be too late

Foster a Quality Culture

- Do things right the first time
- Old saying
  - There is never time to do it right, but always time to do it over
    - Systems are too large and critical
    - Costs of rework and downtime
    - Rework
    - Perpetuates fire fighting cycle

Foster a Quality Culture

- When fixing a defect, think about the problem, not just the symptom
  - Get away from “find and fix”
  - Keep track of the defects you generate
  - When fixing a defect, ask yourself if you may have done the same thing elsewhere
  - If you find a common defect, fix it now; don’t wait for the system to break

Foster a Quality Culture

- Think about maintenance
  - Forget slick solutions that are confusing
  - Follow a consistent coding style
  - Evaluate the complexity of your code
    - Nesting levels
    - Execution path
    - Module sizes
    - Number of subprogram calls
    - Number of variables declared; passed to subprograms

Foster a Quality Culture

- Think about possible reuse
  - Use of globals
  - Hardcoded values (UGH!)
  - Module size
  - Loose coupling and tight cohesion
  - Understandability

Foster a Quality Culture

- Don’t rely on compiler or debugger to find errors
  - Scenario
    - On your first attempt to compile a module, you generate several compiler errors
    - Why should this be of concern?
  - DO pay attention to the compiler or debugger; never ignore or suppress warnings

Foster a Quality Culture

- Think about prevention, not detection
- Do things right the first time
- When fixing a defect, think about the problem, not just the symptom
- Think about maintenance
- Think about possible reuse
- Don’t rely on the compiler or a debugger to find errors

Strings

- Several issues with C-type null terminated strings
  - Is the Null terminator set?
  - Does the size include the null terminator?
  - When accessed, is the range correct?
  - In C, this is not an object

String Functions

- As you saw in the reading, there are many vulnerabilities with strings using existing libraries
- If strings are involved (passed to, returned):
  - Check the size
  - Account for the Null terminator
  - Look at input functions that may or may not preserve the Null terminator or check lengths

Example

- char password [12];
- gets (password);
- if (!strcmp (password, “good pass”)) …

- Flaw here is with how the string was read, not necessarily with the string functions themselves

Overflow

- The previous slide demonstrated an overflow
- gets did not check the length; if the user provides > 11 characters, the assignment is still executed and no errors or warnings are generated
- The result is an overflow
  - We will see several examples of overflow
  - Here, we moved the user's input to "password"
  - More about other overflows later

Is this a valid fix?

- char password [12];
- printf (“Enter an 11 digit password \n”);
- gets (password);
- if (!strcmp (password, “good pass”)) …

- If password entered exceeds 11 characters
  - Null terminator not written in correct location
  - Bytes adjacent to password will be overwritten
    - If the adjacent addresses are not in a valid address space, segmentation fault
    - If adjacent addresses are in a valid address space, execution continues with corrupt data

Stack Overflow

- A process loads the code (often called text), data, heap, bss and stack segments
- bss is "block started by symbols" and contains all uninitialized static and global variables
- Segmented memory supports additional security because we can flag read only segments such as the code
- You should know about these segments from your earlier courses

Example

```
static int GLOBAL_INIT = 1; // data segment, global static
int global_uninit; // BSS segment, global
void myFunct (int number, char *names[]) { // stack, local
  int local_init = 1; // stack, local
  int local_uninit; // stack, local
  static int local_static_init = 1; // data seg, local
  static int local_static_uninit; // BSS segment, local

  // below, storage for buff_ptr is stack, local  but allocated memory is on the heap, local
  int *buff_ptr = (int *)malloc(32);
}
```

- Source: Adapted from Professor Howles

Stack Smashing

- Occurs when the runtime stack is unintentionally modified (due to a defect), or intentionally modified (by an attacker) during the program's execution
  - May result in corrupted local variables
  - An injected address at the top of the stack can change the control flow of the program

Two Examples

- Arc injection, also called return-into-libc
  - This is the example where the same program was executed, but the hacker branched around the password check
- Code injection
  - The example where he injected code to execute the calendar program

C++

- Several libraries available to support strings
  - Be sure you understand limitations of each
  - May reference as a pointer
  - May allow [ ] subscripts but not necessarily perform bounds checking
  - May provide different ways to access contents, determine if empty, or return length
- Safe string libraries in C

How do we mitigate the risk of overflows?

- Always validate all inputs
- Look at string library functions
  - strcpy – copied one string to another
  - strncpy – copies "n" bytes of a string to another
  - Does strncpy sound fail-safe?
  - TR 24731 strcpy_s
    - Only safe if you check return values!

Many other string functions are flawed

Mitigate Risk of Overflows [1]

- Rework legacy code
- Check bounds
- Watch off-by-one errors
- Use of X++ or ++X
- Use safe languages
- Make use of static code analyzers and other tools

Mitigate Risk of Overflows [2]

- Leverage hardware and operating system support to disallow execution off the stack
  - AMD and Intel x86-64 chips with associated 64-bit operating systems
  - Windows XP SP2 (both 32- and 64-bit)
  - Windows 2003 SP1 (both 32- and 64-bit)
  - Linux after 2.6.8 on AMD and x86-64 processors in 32- and 64-bit mode
  - OpenBSD (w^x on Intel, AMD, SPARC, Alpha and PowerPC)
  - Solaris 2.6 and later with the "noexec_user_stack" flag enabled

Mitigate Risk of Overflows [3]

- Look for CPU features to disallow execution of code in a stack section of memory
  - Sun’s SPARC chips had this feature; recent Solaris versions enable it
    - Other operating systems and hardware?
  - Can still allow a modification of the return address, but an exception will be generated
- Also AMD and Intel x86 chips with NX feature
  - Added a new bit to the page table to mark as non-executable

Number Manipulation

- C/C++ code is prone to many flaws
  - Sign errors
  - Overflow
  - Loss of precision; truncation
- Easy to introduce flaws
  - Declaring inadequate types
  - Assignments (compiler may catch some issues)
  - Results of math operations

Numeric Types

- Need to understand
  - Limitations of numeric types
  - What is considered an error
  - Example
    - Overflow with unsigned integers is not considered a problem because of how unsigned ints are defined (modulo)
    - Short unsigned integers roll back to 0
      - This was how it was intended to work

Range and Type Checking

- In C/C++, primarily up to programmer
- C's gcc compiler provides limited capabilities
  - Declare a short int = 32700, multiply * 2 and print result
    - No compiler errors or warnings; lint does not flag any errors.

Options?

- Adopt specially designed safer libraries
- Cast to larger size variable before applying operator; check result and cast back
- Adopt and strictly enforce designs by contract
- Pre conditions
  - Values, signs, magnitude; check types
  - Check for attempts to execute invalid operations (divide by zero, square root of a negative number).

Options? (2)

- Post conditions
    - Are results within limits?
    - Use shifts to verify; assertions
- Code reviews and static analysis are particularly helpful for detecting these errors
- Be very careful with pointers, especially any math operations on pointers!

Arrays

- Vulnerable in C/C++ because
  - No bounds checking
  - Easy off-by-one mistakes; 0 indexing
  - Basis for traditional strings
- Similar problems already discussed
  - Must verify bounds
  - Subject to overflows
  - Can easily allow the overwriting of, or access to data beyond the end of the array (depending on addresses)

More Secure Coding to Come