Defenses

Philip W. L. Fong

Department of Computer Science
University of Calgary
Calgary, Alberta, Canada

CPSC 525/625 (Fall 2016)
Outline

1. Defenses for Backward Compatibility
2. Safe Programming Language
3. Rethinking the Software Development Process
Non-executable Stack

- Stack-based buffer overrun:
  - Inject shellcode into stack buffers
  - Direct control flow to jump to shellcode
Non-executable Stack

- Stack-based buffer overrun:
  - Inject shellcode into stack buffers
  - Direct control flow to jump to shellcode

- Solution idea: make stack non-executable
Non-executable Stack

- Stack-based buffer overrun:
  - Inject shellcode into stack buffers
  - Direct control flow to jump to shellcode
- Solution idea: make stack non-executable
- It does not prevent:
  - Heap smashing
Non-executable Stack

- Stack-based buffer overrun:
  - Inject shellcode into stack buffers
  - Direct control flow to jump to shellcode
- Solution idea: make stack non-executable
- It does not prevent:
  - Heap smashing
    - How about we make writable memory non-executable?
Non-executable Stack

- Stack-based buffer overrun:
  - Inject shellcode into stack buffers
  - Direct control flow to jump to shellcode

- Solution idea: make stack non-executable

- It does not prevent:
  - Heap smashing
    - How about we make writable memory non-executable?
  - Return-to-libc
Non-executable Stack

- Stack-based buffer overrun:
  - Inject shellcode into stack buffers
  - Direct control flow to jump to shellcode
- Solution idea: make stack non-executable
- It does not prevent:
  - Heap smashing
    - How about we make writable memory non-executable?
  - Return-to-libc
  - Stack-based buffer overrun that does not aim at manipulating control flow
Canaries

- Idea: (Figure from [Gollmann])
  - Put a known value (canary) around the return address in the stack frame.
  - Check that the value is intact before returning.
- Can be adapted for the heap.

Figure 10.6: Canary Indicating an Attempted Buffer Overrun Attack on the Stack
**Address Space Randomization (ASR)**

**Idea:**
- Shuffles the locations of variables and other static data stored on the stack.
- Harder to craft an exploit that works all the time.
Code Analyzer

- **Static program analysis**
  - Analyze the “text” of the code without running it.
Code Analyzer

- **Static program analysis**
  - Analyze the “text” of the code without running it.

- **Example applications:**
  - Buffer overflow
  - Format string errors
  - TOCTOU
  - Taint analysis (e.g., SQL injection)
Code Analyzer

- **Static program analysis**
  - Analyze the “text” of the code without running it.

- **Example applications**:
  - Buffer overflow
  - Format string errors
  - TOCTOU
  - Taint analysis (e.g., SQL injection)

- **Problem**: Lots of false positives
  - Usually too conservative
  - Discussion: Why?
Outline

1. Defenses for Backward Compatibility
2. Safe Programming Language
3. Rethinking the Software Development Process
Discussion

- What peculiarities of C/C++ make programs written in them usual targets for attacks?
Memory Safety

- Automated memory management: aka garbage collection
  - No more `malloc` and `free`
- All buffer accesses are checked for boundary errors.
- Code cannot be overwritten.
- Data cannot be executed.
- The runtime stack cannot be manipulated.
Type Safety

Definition (Type Safety)

A value of a certain type is only operated on by operations supported by that type.

- Pointers cannot be forged
- No type confusion
- Encapsulation (access control via public/private) cannot be penetrated.
Control Flow Integrity

- Control is directed only to legitimate program points.
- Function/method returns will only return control to the caller.
- Cannot forge an address and jump to it.
Discussion

- What are some safe languages that you are aware of?
  - i.e., with memory safety, type safety, and control flow integrity
What are some safe languages that you are aware of?
- i.e., with memory safety, type safety, and control flow integrity

Let’s imagine carrying out a buffer overrun attack in a safe language, and see how each of the above features prevent the execution of the exploit.
Discussion

- What is stopping people from building “real software” (think Apache) using a safe language such as Haskell?
Haskell for System Programming (Really?)

- Haskell for OS and Systems Programming
  http://wiki.haskell.org/Applications_and_libraries/Operating_system
- Ivory
  http://ivorylang.org/
- Kinetic
  http://intoverflow.wordpress.com/kinetic/
- House
  http://programatica.cs.pdx.edu/House/
Some Cool Research

- Proof-Carrying Code
- Typed Assembly Language
Outline

1. Defenses for Backward Compatibility
2. Safe Programming Language
3. Rethinking the Software Development Process
Introducing Security Considerations into the Process

1. Requirement
2. Design
3. Implementation
4. Testing
Introducing Security Considerations into the Process

1. **Requirement**
   - Aim to craft a small product: easy to validate

2. **Design**

3. **Implementation**

4. **Testing**
Introducing Security Considerations into the Process

1. **Requirement**
   - Aim to craft a small product: easy to validate

2. **Design**
   - Threat modeling

3. **Implementation**

4. **Testing**
Introducing Security Considerations into the Process

1. **Requirement**
   - Aim to craft a small product: easy to validate

2. **Design**
   - Threat modeling

3. **Implementation**
   - No longer optimizing for time-to-market
   - Coding practices that improve robustness of software

4. **Testing**
Introducing Security Considerations into the Process

1. **Requirement**
   - Aim to craft a small product: easy to validate

2. **Design**
   - Threat modeling

3. **Implementation**
   - No longer optimizing for time-to-market
   - Coding practices that improve robustness of software

4. **Testing**
   - Fuzz testing
   - Static analysis
Most of the materials in these slides are based on:

- [Smith & Marchesini], §6.1–6.6