Secure coding in C and C++ - ARM

CYDCp3d_ARM | 3 days | On-site or online | Hands-on

Your application written in C and C++ works as intended, so you are done, right? But did you consider feeding in incorrect values? 16Gbs of data? A null? An apostrophe? Negative numbers, or specifically -1 or -2^31? Because that's what the bad guys will do – and the list is far from complete.

Handling security needs a healthy level of paranoia, and this is what this course provides: a strong emotional engagement by lots of hands-on labs and stories from real life, all to substantially improve code hygiene. Mistakes, consequences, and best practices are our blood, sweat and tears.

All this is put in the context of C and C++, and extended by core programming issues, discussing security pitfalls of these languages.

So that you are prepared for the forces of the dark side.

So that nothing unexpected happens.

Nothing.

Cyber security skills and drills


28 LABS


5 CASE STUDIES

Audience

C/C++ developers

Group size

12 participants

Preparedness

General C/C++ development

Outline

- Cyber security basics
- Memory management vulnerabilities
- Memory management hardening
- Common software security weaknesses
- Wrap up

Standards and references

SEI CERT, CWE and Fortify Taxonomy

What you'll have learned

- Getting familiar with essential cyber security concepts
- Identify vulnerabilities and their consequences
- Learn the security best practices in C and C++
- Input validation approaches and principles

Table of contents

Day 1

Best practices and some typical mistakes

- Unsafe functions
- Dealing with unsafe functions

Lab – Fixing buffer overflow

- What's the problem with asctime()?

Lab – The problem with asctime()

- Using std::string in C++
- Unterminated strings
- readlink() and string termination
- Manipulating C-style strings in C++
- Malicious string termination

Lab – String termination confusion

- String length calculation mistakes
- Off-by-one errors
- Allocating nothing

Day 2

› Memory management hardening

Securing the toolchain

- Securing the toolchain in C and C++
- Using FORTIFY_SOURCE

Lab – Effects of FORTIFY

- AddressSanitizer (ASan)
  - Using AddressSanitizer (ASan)

  Lab – Using AddressSanitizer
- Stack smashing protection
  - Detecting BoF with a stack canary
  - Argument cloning
  - Stack smashing protection on various platforms
  - SSP changes to the prologue and epilogue

  Lab – Effects of stack smashing protection

Runtime protections

- Runtime instrumentation
- Address Space Layout Randomization (ASLR)
  - ASLR on various platforms

- *Lab – Effects of ASLR*
  - Circumventing ASLR – NOP sleds
  - Circumventing ASLR – memory leakage
- Non-executable memory areas
  - The NX bit
  - Write XOR Execute (W^X)
  - NX on various platforms
  - *Lab – Effects of NX*
  - NX circumvention – Code reuse attacks
    - Return-to-libc / arc injection
  - Return Oriented Programming (ROP)
    - Protection against ROP
    - ARM-specific ROP protection techniques

› Common software security weaknesses

Security features

- Authentication
  - Authentication basics
  - Multi-factor authentication
- Password management
  - Inbound password management
    - Storing account passwords
    - Password in transit
    - *Lab – Is just hashing passwords enough?*
    - Dictionary attacks and brute forcing
    - Salting
    - Adaptive hash functions for password storage
    - Password policy
    - NIST authenticator requirements for memorized secrets
      - *Case study – The Ashley Madison data breach*
      - *The dictionary attack*
      - *The ultimate crack*
      - *Exploitation and the lessons learned*

Code quality

- Code quality and security
- Data handling
  - Type mismatch
  - *Lab – Type mismatch*
  - Initialization and cleanup
    - Constructors and destructors
    - Initialization of static objects

- *Lab – Initialization cycles*
- Array disposal in C++
- *Lab – Mixing delete and delete[]*

- Memory and pointers
  - Memory and pointer issues
  - Pointer handling pitfalls
  - Pointer usage in C and C++
    - Use after free
    - *Lab – Use after free*
    - *Lab – Runtime instrumentation*
    - Double free
    - Memory leak
    - Smart pointers and RAII
    - Smart pointer challenges

Day 3

› Common software security weaknesses

Input validation

- Input validation principles
- Denylists and allowlists
- What to validate – the attack surface
- Where to validate – defense in depth
- When to validate – validation vs transformations
- Validation with regex
- Regular expression denial of service (ReDoS)

Lab – ReDoS in C

- Dealing with ReDoS
- Injection
  - Injection principles
  - Injection attacks
  - Code injection
    - OS command injection
    - *Lab – Command injection*
    - OS command injection best practices
    - Avoiding command injection with the right APIs
    - *Lab – Command injection best practices*
      - *Case study – Shellshock*
    - *Lab - Shellshock*

- Process control – library injection
  - Library hijacking
  - *Lab – Library hijacking*
- Integer handling problems
  - Representing signed numbers
  - Integer visualization
  - Integer promotion
  - Integer overflow
  - *Lab – Integer overflow*
  - Signed / unsigned confusion
  - *Case study – The Stockholm Stock Exchange*
  - *Lab – Signed / unsigned confusion*
  - Integer truncation
  - *Lab – Integer truncation*
  - *Case study – WannaCry*
  - Best practices
    - Upcasting
    - Precondition testing
    - Postcondition testing
    - Best practices in C
    - *Lab – Handling integer overflow on the toolchain level in C and C++*
    - Best practices in C++
    - *Lab – Integer handling best practices in C++*
- Files and streams
  - Path traversal
  - *Lab – Path traversal*
  - Path traversal-related examples
  - Path traversal best practices
  - *Lab – Path canonicalization*

› Wrap up

Secure coding principles

- Principles of robust programming by Matt Bishop
- Secure design principles of Saltzer and Schroeder

And now what?

- Software security sources and further reading
- C and C++ resources