Secure coding in C and C++ for medical devices

CYDCp_MedDev | 4 days | Hands-on

Your medical device 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.

The most important concern in the healthcare industry is naturally safety. However, once isolated medical devices became highly connected to date, which poses new kinds of security risks: from exposing sensitive patient information to denial of service. And remember, there is no safety without security!

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 medical devices developed in 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.

Audience

C/C++ developers developing medical devices

Group size

12 participants

Preparedness

General C/C++ development

Platforms

Linux

Windows

Labs

Hands-on

Outline

- Cyber security basics
- Buffer overflow
- Memory management hardening
- Common software security weaknesses
- Using vulnerable components
- Security testing
- Wrap up

Objective list

- Getting familiar with essential cyber security concepts
- Learning about security specialties of the healthcare sector
- Handling security challenges in your C and C++ code
- Identify vulnerabilities and their consequences
- Learn the security best practices in C and C++
- Understanding security testing methodology and approaches
- Getting familiar with common security testing techniques and tools

Table of contents

Day 1

- Calling conventions on x64
- The stack frame
- Stacked function calls

Memory management vulnerabilities

- Memory management and security
- Vulnerabilities in the real world
- Buffer security issues
- Buffer overflow on the stack
  - Buffer overflow on the stack – stack smashing
  - Exploitation – Hijacking the control flow

  Lab – Buffer overflow 101, code reuse
  - Exploitation – Arbitrary code execution
  - Injecting shellcode

  Lab – Code injection, exploitation with shellcode

  Case study – Stack BOF in boot file handling of MQX DHCP client
- Buffer overflow on the heap
  - Unsafe unlinking

  Case study – Heap BOF in VxWorks DHCP options parsing

  Case study – Heartbleed
- Pointer manipulation
  - Modification of jump tables
  - Overwriting function pointers

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++

Day 2

› Buffer overflow

Some typical mistakes leading to BOF

- 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

Case study – Off-by-one error in VxWorks TCP 'Urgent Data' parsing

- Allocating nothing

› Memory management hardening

Securing the toolchain

- Securing the toolchain in C and C++
- Compiler warnings and security
- Using FORTIFY_SOURCE

Lab – Effects of FORTIFY

- AddressSanitizer (ASan)
  - Using AddressSanitizer (ASan)
  - ASan changes to the prologue
  - ASan changes to memory read/write operations
  - ASan changes to the epilogue

  Lab – Using AddressSanitizer
- RELRO protection against GOT hijacking
- Heap overflow protection
- 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
  - Bypassing stack smashing protection

Runtime protections

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

  Lab – Effects of ASLR
  - Circumventing ASLR – NOP sleds
  - Heap spraying
- 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)

Lab – ROP demonstration

  - Protection against ROP

› Common software security weaknesses

Security features

- Authentication
  - Authentication basics
  - Authentication weaknesses

Case study – Missing authentication in Alaris TIVA

  - User interface best practices
- 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
    - Password length
    - Password hardening
    - Using passphrases

Case study – The Ashley Madison data breach

The dictionary attack

The ultimate crack

Exploitation and the lessons learned

    - Password database migration
- Authorization
  - Access control basics

Case study – Broken authorization in Conexus protocol for Medtronic devices

  - File system access control
    - Improper file system access control
    - Ownership
    - chroot jail

- Using umask()
- Linux filesystem
- LDAP

Case study – Insecure file permissions in McKesson Cardiology 13.x / 14.x

Day 3

› Common software security weaknesses

Security features

- Authentication
- Password management
  - Outbound password management
    - Hard coded passwords
    - Best practices

Lab – Hardcoded password

Case study – Compromising Abbott FreeStyle Libre sensors via NFC

- Protecting sensitive information in memory
- Challenges in protecting memory
- Heap inspection
- Compiler optimization challenges

Lab – Zeroization challenges

- Sensitive info in non-locked memory

› Common software security weaknesses

Input validation

- Input validation principles
  - Blacklists and whitelists
  - Data validation techniques

Case study – Missing input validation in Natus Xltek NeuroWorks 8

  - What to validate – the attack surface
  - Where to validate – defense in depth
  - How to validate – validation vs transformations
  - Output sanitization
  - Encoding challenges
  - Validation with regex
- 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*
- *Case study – Command injection in GE Healthcare MobileLink*
- Process control – library injection
- DLL hijacking
- *Lab – DLL hijacking*
- *Case study – DLL injection in Vyaire Medical CareFusion Upgrade Utility*

- Integer handling problems
  - Representing signed numbers
  - Integer visualization
  - Integer promotion
  - Integer overflow
  - *Lab – Integer overflow*
  - Signed / unsigned confusion
  - *Lab – Signed / unsigned confusion*
  - Integer truncation
  - *Lab – Integer truncation*
  - *Case study – WannaCry*
  - Best practices
    - Upcasting
    - Precondition testing
    - Postcondition testing
    - Using big integer libraries
    - Best practices in C
    - UBSan changes to arithmetics
    - *Lab – Handling integer overflow on the toolchain level in C/C++*
    - Best practices in C++
    - *Lab – Integer handling best practices in C++*
- Files and streams
  - Path traversal
  - Path traversal-related examples
  - *Lab – Path traversal*
  - Path traversal best practices
  - *Lab – Path canonicalization*
- Format string issues
  - The problem with printf()
  - *Lab – Exploiting format string*

Day 4

› Common software security weaknesses

Time and state

- Race conditions
  - Race condition in object data members
    - *Case study – State confusion in VxWorks IPNet stack*
  - File race condition
    - *Lab - TOCTTOU*
    - Insecure temporary file
  - Potential race conditions in C/C++
    - Race condition in signal handling
    - Forking
    - Bit-field access

Errors

- Error and exception handling principles
- Error handling
  - Returning a misleading status code
  - Error handling in C
  - Error handling in C++
  - Using std::optional safely
  - Information exposure through error reporting
- Exception handling
  - In the catch block. And now what?
  - Empty catch block
  - Exception handling in C++

  Lab – Exception handling mess

Code quality

- Data
  - Type mismatch

  Lab – Type mismatch
  - Initialization and cleanup
    - Constructors and destructors
    - Initialization of static objects

    Lab – Initialization cycles
  - Unreleased resource

    Case study – Unreleased resource in VxWorks TCP 'Urgent Data' parsing
    - Array disposal in C++

    Lab – Mixing delete and delete[]

- Control flow
  - Incorrect block delimitation
  - Dead code
  - Leftover debug code
  - Backdoors, dev functions and other undocumented functions
  - Using if-then-else and switch defensively
- Signal handling
  - Signal handlers
  - Best practices
- Object oriented programming pitfalls
  - Inheritance and object slicing
  - Implementing the copy operator
  - The copy operator and mutability
  - Mutability
    - Mutable predicate function objects
    - *Lab – Mutable predicate function object*
- Memory and pointers
  - Memory and pointer issues
  - Pointer handling pitfalls
  - Alignment
  - Null pointers
    - NULL dereference
    - NULL dereference in pointer-to-member operators
    - *Case study – NULL dereference in VxWorks IGMP parsing*
  - 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
    - Incorrect pointer arithmetics
- File I/O
  - Working with file descriptors, structures and objects
  - File reading and writing
  - File access functions and methods

› Using vulnerable components

Assessing the environment
Hardening
Case study – Supply chain attack on Alaris Gateway Workstation

Vulnerability management

- Patch management
- Vulnerability management
- Vulnerability databases

Lab – Finding vulnerabilities in third-party components

- DevOps, the build process and CI / CD
- Insecure compiler optimization

› Security testing

Security testing vs functional testing
Manual and automated methods

Security testing techniques and tools

- Code analysis
  - Security aspects of code review
  - Static Application Security Testing (SAST)

  Lab – Using static analysis tools
- Dynamic analysis
  - Security testing at runtime
  - Penetration testing
  - Stress testing
  - Dynamic analysis tools
  - Dynamic Application Security Testing (DAST)
  - Fuzzing
  - Fuzzing techniques
  - Fuzzing – Observing the process

› Wrap up

Secure coding principles

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

And now what?

- Software security sources and further reading

- C and C++ resources