Secure coding in C and C++ for automotive

CYDCp_Auto | 4 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.

To date vehicles become highly connected - not only between the internal components, but also to the outside worlds. Todays cars are already running millions of lines of source code, and this introduces a new set of risks to the industry that is historically concerned about safety. Even though some of the attacks are still theoretical, many of the standards already started introducing security considerations.

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.

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

So that nothing unexpected happens.

Nothing.

Cyber security skills and drills

32 LABS

12 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

MISRA, SEI CERT, CWE and Fortify Taxonomy

What you'll have learned

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

Table of contents

Day 1

- Vulnerabilities in the real world
- Mapping to MISRA
- 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 the Wi-Fi stack of the Tesla Model S*
- Buffer overflow on the heap
  - Unsafe unlinking
  - *Case study – Heartbleed*
- Pointer manipulation
  - Modification of jump tables
  - GOT and PLT
  - Overwriting function pointers

Day 2

› Memory management vulnerabilities

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++

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

- 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 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
  - Circumventing ASLR – memory leakage
  - 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
    - Chained return-to-libc

- Return Oriented Programming (ROP)
  - *Lab – ROP demonstration*
  - Whatever Oriented Programming
  - Protection against ROP
  - ARM-specific ROP protection techniques

› Common software security weaknesses

Security features

- Authentication
  - Authentication basics
  - Multi-factor authentication
  - Authentication weaknesses
  - *Case study – Weak authentication controls for airbag detonation*
  - *Case study – Hacking the Mitsubishi Outlander PHEV hybrid*
- Authorization
  - Access control basics
  - File system access control
    - Improper file system access control
    - Ownership
    - chroot jail
    - Using umask()
    - Hardening the Linux filesystem
    - Lightweight Directory Access Protocol (LDAP)

Day 3

› Common software security weaknesses

Security features (continued)

- 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*
  - Outbound password management
    - Hard coded passwords
    - Best practices
    - *Lab – Hardcoded password*
      - *Case study – Hard-coded credentials in MyCar vehicle control app*

› Common software security weaknesses

Input validation

- Input validation principles
- Denylists and allowlists
- What to validate – the attack surface
- Attack surface in automotive
- Where to validate – defense in depth
- When to validate – validation vs transformations
- Output sanitization
- Encoding challenges
- Unicode challenges
- Validation with regex
- Regular expression denial of service (ReDoS)
- *Lab – ReDoS in C*
- Dealing with ReDoS
- Mapping to MISRA
- Integer handling problems
  - Representing signed numbers
  - Integer visualization
  - The MISRA essential type model
  - Integer promotion
  - Composite expressions and type conversion
  - 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
- *Lab – Handling integer overflow on the toolchain level in C and C++*
- Best practices in C++
- *Lab – Integer handling best practices in C++*

Day 4

› Common software security weaknesses

Input validation

- 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 Jeep Cherokee*
- Process control – library injection
  - Library hijacking
  - *Lab – Library hijacking*
- Files and streams
  - Path traversal
  - *Lab – Path traversal*
  - Path traversal-related examples
  - Virtual resources
  - Path traversal best practices
  - *Lab – Path canonicalization*
- Format string issues
  - The problem with printf()
  - *Lab – Exploiting format string*
  - *Case study – Format string problems in various infotainment software*

Time and state

- Race conditions

- File race condition
  - Time of check to time of usage – TOCTTOU
  - TOCTTOU attacks in practice
  - *Lab - TOCTTOU*
  - Insecure temporary file

Errors

- Error and exception handling principles
- Mapping to MISRA
- 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
  - *Lab – Exception handling mess*

Code quality

- Code quality and security
- Mapping to MISRA
- Data handling
  - Type mismatch
  - *Lab – Type mismatch*
  - Initialization and cleanup
    - Constructors and destructors
    - Initialization of static objects
    - *Lab – Initialization cycles*
  - Unreleased resource
    - Array disposal in C++
    - *Lab – Mixing delete and delete[]*
- 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*

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
- Links to automotive coding standards