

Code responsibly with generative AI in C++

CYDCpp3dCop | 3 days | On-site or online | Hands-on

Embark on a comprehensive exploration of cybersecurity and secure coding practices in this intensive three-day course. It is primarily focusing on C++, but also integrates some C concepts. Based on a primer on machine code, assembly, and memory overlay (Intel and ARM versions available), the curriculum addresses critical security issues related to memory management. Various protection techniques on the level of source code, compiler, OS or hardware are discussed - such as stack smashing protection, ASLR or the non-execution bit - to understand how they work and make clear what we can and what we can't expect from them.

The various secure coding subjects are aligned to common software security weakness categories, such as security features, error handling or code quality. Many of the weaknesses are, however, linked to missing or improper input validation. In this category you'll learn about injection, the surprising world of integer overflows, and about handling file names correctly to avoid path traversal.

Through hands-on labs and real-world case studies, you will navigate the details of secure coding practices to get essential approaches and skills in cybersecurity.

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

So that nothing unexpected happens.

Nothing.

This variant of the course deals extensively with how certain security problems in code are handled by GitHub Copilot.

Through a number of hands-on labs participants will get first hand experience about how to use Copilot responsibly, and how to prompt it to generate the most secure code. In some cases it is trivial, but in most of the cases it is not; and in yet some other cases it is basically impossible.

At the same time, the labs provide general experience with using Copilot in everyday coding practice - what you can expect from it, and what are those areas where you shouldn't rely on it.

Cyber security skills and drills



30 LABS



6 CASE STUDIES

Audience

C/C++ developers

Group size

12 participants

Preparedness

General C++ and 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
- Correctly implementing various security features
- Identify vulnerabilities and their consequences
- Learn the security best practices in C++
- Input validation approaches and principles

Table of contents

Day 1

› Cyber security basics

What is security?

Threat and risk

[Cyber security threat types – the CIA triad](#)

Cyber security threat types – the STRIDE model

Consequences of insecure software

› Memory management vulnerabilities

Assembly basics and calling conventions

- x64 assembly essentials
- Registers and addressing
- Most common instructions
- Calling conventions on x64
 - Calling convention – what it is all about
 - Calling convention on x64
 - The stack frame
 - Stacked function calls

Buffer overflow

- Memory management and security
- 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*
- Buffer overflow on the heap
 - Unsafe unlinking
 - 📖 *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*
- 🔗 *Lab – Experimenting with buffer overflow in Copilot*
- Using `std::string` in C++
- 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++
- 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

› Common software security weaknesses

Security features

- 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 ultimate crack*
 -  *Exploitation and the lessons learned*
 - Password database migration

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*
 - Unreleased resource
 - Array disposal in C++

- 🔗 *Lab – Mixing delete and delete[]*
- Object oriented programming pitfalls
 - Accessibility modifiers
 - Are accessibility modifiers a security feature?
 - Inheritance and object slicing
 - Implementing the copy operator
 - The copy operator and mutability
 - Mutability
 - Mutable predicate function objects
- 🔗 *Lab – Mutable predicate function object*

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
- Injection
 - 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*
 - 🔗 *Lab – Experimenting with command injection in Copilot*
 - 📖 *Case study – Shellshock*
 - 🔗 *Lab – Shellshock*
 - 📖 *Case study – Command injection via ping*
- Integer handling problems
 - Representing signed numbers
 - Integer visualization
 - Integer promotion
 - Integer overflow
 - 🔗 *Lab – Integer overflow*
 - 🔗 *Lab – Experimenting with integer overflow in Copilot*
 - Signed / unsigned confusion

- 📖 *Case study – The Stockholm Stock Exchange*
- 🔗 *Lab – Signed / unsigned confusion*
- 🔗 *Lab – Experimenting with signed / unsigned confusion in Copilot*
 - Integer truncation
- 🔗 *Lab – Integer truncation*
- 🔗 *Lab – Experimenting with integer truncation in Copilot*
- 📖 *Case study – WannaCry*
 - Best practices
 - Upcasting
 - Precondition testing
 - Postcondition testing
 - UBSan changes to arithmetics
 - 🔗 *Lab – Handling integer overflow on the toolchain level in C++*
 - Best practices in C++
 - 🔗 *Lab – Integer handling best practices in C++*
- Files and streams
 - Path traversal
 - 🔗 *Lab – Path traversal*
 - Path traversal best practices
 - 🔗 *Lab – Path canonicalization*
 - 🔗 *Lab – Experimenting with path traversal in Copilot*

› 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