




Secure C++ Coding

1. Introduction and Overview

A top-down view of a wooden desk. In the top right, a portion of a silver laptop is visible, showing keys like 'Q', 'W', 'E', 'A', 'S', 'Z', 'X', 'C', 'V', 'B', 'N', 'M', 'Fn', 'Control', and 'Option'. Below the laptop, a pair of black-rimmed glasses lies horizontally. To the right of the glasses is a white ceramic cup filled with dark coffee, with a yellow handle. In the top center, a small green succulent in a dark pot sits on the desk. In the bottom right, the corner of a black tablet or laptop is visible. The desk surface is made of dark wood with prominent grain patterns.

Module Topics

1. **Class Introduction**
2. Some Basics on Code Security
3. C++ Standards
4. C++ Security Standards
5. Implementation of Secure Coding

About this Class

- We are not going to learn “how to program”
 - The assumption is that you are familiar with C++ coding
 - You may be a programmer or a tester or...?
- The main topics we want to cover are:
 - The basics of secure code: what “secure programming” means
 - Why C++ is inherently insecure – features and language design
 - Understand the CERT Secure coding standard
 - C++ Best Practices and Code Standards
 - Secure code development practices
 - Mitigating risk in code development and design

1-3

This course assumes that you either have a knowledge of C++ from either a programming perspective or a testing (including security testing) perspective.

There will be no time to cover programming basics in class.

Introductions

- Just to find out a few things about you:
 - Your area of expertise: programming, design, QA, etc.
 - Current roles and responsibilities
 - Experience with C++ programming and software security
 - Any specific issues or needs you want addressed

Hands-On Environments


- You will be provided with a virtual machine:
 - It will be accessible continuously for the duration of the class
 - Your VM is a "no consequences" environment
 - *it doesn't matter if you break it, so feel free to experiment*
 - Most of the hands on work is experimenting with the non-compliant and compliant code examples
 - You should have already gotten a test link to ensure you can connect to the lab VM

Materials

- All materials used in class will be available:
 - GitHub repository: [github/exgnosis/SecureCPPCoding](https://github.com/exgnosis/SecureCPPCoding)
 - The repository will be up for 30 days from the end of the course
- You are free to download and share what is there
- The only materials not in the repo are those where
 - A copyright is held by a third party that prohibits distribution; and
 - We do not have permission by that party to share the materials

Class Expectations

- Please keep in mind the following:
- This is your class, I am just the facilitator
 - We can deviate from the plan at any time to pursue topics of interest
 - Feel free to ask questions any time, don't feel you have to wait
 - Feel free to contribute any observations or anything else that is relevant to the material being covered
- This is a virtual environment
 - I can't see you so I will be relying on verbal feedback
 - I will be regularly asking question of all of you to ensure we are on track and on the same page

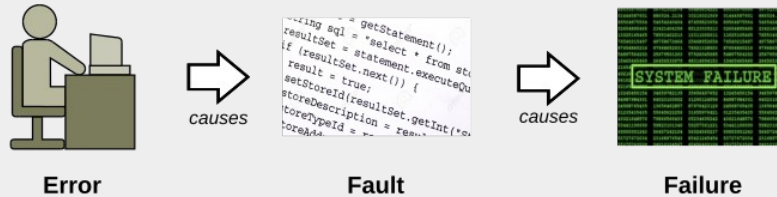


Module Topics

1. Class Introduction
2. **Some Basics on Code Security**
3. C++ Standards
4. C++ Security Standards
5. Implementation of Secure Coding

Some Basic Definitions

Defects



Error: a human action that eventually leads to a fault

Fault: an incorrect step in building the system at any point that results in failure

Failure: any place the software does not perform as required

Defect: a generic term for any of the above

1-9

This terminology is standard in software quality engineering. The term *defect* is often used incorrectly as a synonym for *fault*. However, a defect can also be an error which could occur anywhere in the development process.

For example, missing a critical security requirement is an error, but the fault that it leads to is a missing feature. However, because the testing of the system is based on the erroneous requirements, there are no tests related to the missing requirement which means the tests all pass. However once the application is deployed, the missing feature may now be a gaping security hole that is exploitable by an attacker.

This is what actually meant by a fault, it is not just existing code that is broken but also includes code that should be there but isn't. These faults by omission usually result from missing security requirements which means that they are not picked up during standard testing. Hopefully, they are identified during acceptance testing, security testing or beta testing, but unfortunately many are not picked up until an attack actually occurs in the operational environment.

Relevance to Security

- Error:
 - Not understanding why certain code or designs are risky
- Fault:
 - The error leads to writing exploitable code.
 - Because the risk is not recognized it is not identified during testing
- Failure:
 - The system is deployed with a security flaw
- We are focusing on the errors made by a developer w.r.t.:
 - How they choose to design code
 - How they chose to structure code

1-10

At a most fundamental level, we can trace back many of the security flaws we find to root cause errors like:

- 1) Not understanding how certain code create vulnerabilities
- 2) Not understanding what to look for in code inspections and reviews
- 3) Not understanding how to evaluate code for vulnerabilities
- 4) Not understanding how to test code from a security perspective.

Robust Software

- Software is robust when it has
 - *"the ability to cope with errors during execution and to handle erroneous input"*
- Three types of robustness
 - **Safe**: when the system can detect, respond to or prevent *accidental* harm
 - **Secure**: when the system can detect, respond to or prevent *intentional* harm
 - **Survivable**: when the system is both safe and secure

1-11

The reason we distinguish between the first two kinds of robustness is because we design for them in different ways.

One of the more interesting analogies made by one security tester is that safety testing is about doing a standard logical analysis of what could go wrong while security testing is more like developing models based on game theory where strategies and actions are continuously changing in a highly interactive manner.

Robustness is survivability. In order to be robust, both safety and security have to be addressed but require very different approaches and strategies.

The important point is that while many defects can result in security flaws, the two are not equivalent. Some defects are not security flaws because they cannot be exploited by an attacker, while some features, working exactly as specified, may present security flaws because they are a point where the system is vulnerable to attack.

It is important to distinguish between a security flaw, which is a property of the software, and a vulnerability which is a combination of a security flaw plus the opportunity in an operational environment for that flaw to be exploited by an attacker.

Software Engineering

- Focuses on eliminating defects
 - To remove any faults that prevent the software from working as specified
 - To ensure the software handles the normal and reasonable situations and inputs correctly, including invalid inputs
- Does not focus on intentional attacks
 - Attacks usually involve attempting to put the system into an abnormal situation or unusual state
 - Attacks also usually involve bizarre, unreasonable and highly unusual inputs

1-12

Software engineering is focused on safety and ensuring that software works in a manner that meets the functional and non-functional requirements of the stakeholders, and that the software can be build in a timely and cost-effective manner. The focus of software engineering is on the performance of the software in a normal, expected environment and on the ability of the software to respond to anticipated possible deviations from a normal environment; for example hardware failures or corrupt input data.

However, attacks often begin with putting the software in an abnormal or unusual state that might not be an anticipated deviation from the normal operating state. The system may be bombarded with highly abnormal inputs to put the system into an abnormal state and then to exploit the system while it is in that abnormal state. These sorts of attacks are not usually predictable based on a normal safety analysis of the system.

Another form of attack that often occurs is a probing attack where the attacker tries looking for the places in the system where there exist security flaw which are the result of a defect introduced by an error of omission as discussed in a previous slide. Because this is a defect by omission, normal quality engineering may miss identifying the security flaw introduced by the defect.

Security Engineering

- A **security flaw**:
 - Is a defect in or a feature of the software that can be exploited by an attacker
 - *A defect that is fixed for normal operations may still be a security flaw*
 - Not all defects are security flaws
 - Only defects that can be exploited are security flaws
 - Not all security flaws are the result of defects but they are the result of some form of error
- A **vulnerability**:
 - Is a set of circumstances that allow an attacker to exploit a security flaw
 - Vulnerability = security flaw + exploitation opportunity

1-13

The important point is that while many defects can result in security flaws, the two are not equivalent.

Some defects are not security flaws because they cannot be exploited by an attacker, while some features, working exactly as specified, may present security flaws because they are a point where the system is vulnerable to attack.

It is important to distinguish between a security flaw, which is a property of the software, and a vulnerability which is a combination of a security flaw plus the opportunity in an operational environment for that flaw to be exploited by an attacker.

Security Engineering

- A *mitigation* is the removal of a vulnerability either
 - By fixing the underlying security flaw; or
 - Developing a *workaround* that prevents attackers from accessing the security flaw
- Not all security flaws can be fixed
 - The cost of fixing the flaw may be prohibitive
 - The flaw may be complex or involve multiple components which means it may be a systemic problem not a defect

1-14

A mitigation removes a vulnerability.

There are two ways to mitigate a vulnerability


- 1) Fix the security flaw
- 2) remove the opportunity for an attacker to access the flaw.

In some cases, fixing a security flaw is impossible. The flaw may be an artifact of the overall design, too expensive to fix or is actually a critical feature of the software that can be exploited, as opposed to being a defect that can be fixed. In these cases, the mitigation involves eliminating the vulnerability by preventing the flaw from being accessed.

There are a variety of terms for this, the most common of which is called a workaround, but this usually involve establishing an intermediate layer that either filters access to the flaw or wraps the component with the flaw in a new interface that hides the flaw.

Security Engineering

- Summary:
 - Not all defects are security flaws
 - Not all security flaws are defects
 - A defect is not a security flaw if it cannot be exploited
 - A vulnerability is a situation where an attacker can exploit a security flaw
 - A mitigation is the the removal of a vulnerability
 - Any defect has the potential to be a security flaw

A top-down view of a wooden desk. In the top right, a portion of a silver laptop is visible, showing keys like 'Q', 'W', 'E', 'A', 'S', 'Z', 'X', 'C', 'V', 'B', 'N', 'M', 'comma', 'period', 'backslash', 'forward slash', 'asterisk', 'hash', 'underscore', 'bracket', 'parenthesis', 'at', 'number', 'zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'equals', 'plus', 'minus', 'multiply', 'divide', 'less than', 'greater than', 'less than or equal to', 'greater than or equal to', 'percent', 'dollar sign', 'pound sign', 'yen sign', 'euro sign', 'dollar sign', 'pound sign', 'yen sign', 'euro sign'. In the center, a pair of black-rimmed glasses lies horizontally. To the right of the glasses is a white ceramic cup filled with dark coffee, with a yellow handle. In the top center, a small green succulent in a dark pot sits. In the bottom right, a portion of a black tablet is visible, showing a small white circle in the center of its back.

Module Topics

1. Introductions to Best Practices
2. Organizational Issues
3. **C++ Standards**
4. C++ Security Standards
5. Implementation of Secure Coding

C++ Language Standard

- Maintained by an ISO working group
 - Multiple releases identified by year: eg C++17 in 2017
 - New standards generally include specifications for new features and libraries
- Secure programming is not part of the standard
 - The standard defines behavior and structures that are important for inter-operability and portability between platforms
 - The standard ensures consistent treatment of the language by different compilers
 - Code written to the standard should compile and produce the same functional result on any C++ compliant compiler

1-17

There is a common misconception that if C++ code is written so that it is standards compliant, then it must be secure. The problem is that the C++ language standard, like any language standard, is not concerned with the behavior of programs written in that language or their security – those are not the goals of a language standard.

The primary goals of a language standard is standardization which:

- 1) Provides guidance to different compiler authors what constitutes the well-formed C++ code that their compiler is required to accept as input.
- 2) Ensures that the functional component produced by a compiler for a specific C++ construct is consistent across compilers.
- 3) Ensures that code written to the standard is portable and will compile and produce the same functional executable from any standards compliant compiler. In this sense, the executable output from a compiler is considered a black box which may have different internal structures depending on the compiler which produced it, but the black box will behave the same if that behavior is defined by the standard.

There is no part of the standard that prohibits any compiler developer from adding features to their version of C++ as long as those extensions do not compromise or contradict what is described in the standard.

C++ Language Standard

- Not all behavior of C++ code is defined
- The standard allows for the following behaviors:
 - *Unspecified*: when unspecified values are used or no specification of which one of multiple options should be used
 - *Locale-specific*: depends on local conventions like character set collation order and date formats
 - *Implementation-defined*: allows an implementation to decide how to implement the behavior provided it is documented
 - *Undefined*: what happens when erroneous inputs or incorrect program constructs are used
- Undefined behaviors are a source of security flaws

1-18

The important point to note here is that not all of the behavior of C++ expression is defined because it is not required to be in all cases.

- 1) **Locale-specific behavior**: these are variations due to language, culture, etc. For example, whether the `islower()` function returns true for characters other than the 26 lowercase Latin letters is locale specific or what is considered legal formatting for a date or time string.
- 2) **Unspecified behavior**: the specification does not describe the result of using of an unspecified value, or the behavior where the standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance. An example of this would be the order in which the arguments to a function are evaluated.
- 3) **Implementation-defined behavior**: these are places where each compiler implementation can decide on the behavior but documents how the choice is made. These sorts of decisions are often left to the compiler in places where they would depend on underlying hardware and operating system architectures.
- 4) **Undefined behavior**: what happens when the program construct is used incorrectly or with incorrect inputs.

C Language Standard

- C++ was originally a super set of C
 - C++ code was translated into C code then compiled
 - Generally true at that time that valid C code was also valid C++
- C also has an ISO standard
 - The C++ Standard depends on the C standard
 - There are points of variation between the two standards
- Plus each standard has different releases
 - Always a question of cross language AND backwards compatibility
 - eg. Should C++ support a new feature of C?
- We will not be concerned with this issue in this course

1-19

The important point to note here is that not all of the behavior of C++ expression is defined because it is not required to be in all cases.

- 1) Locale-specific behavior: these are variations due to language, culture, etc. For example, whether the `islower()` function returns true for characters other than the 26 lowercase Latin letters is locale specific or what is considered legal formatting for a date or time string.
- 2) Unspecified behavior: the specification does not describe the result of using of an unspecified value, or the behavior where the standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance. An example of this would be the order in which the arguments to a function are evaluated.
- 3) Implementation-defined behavior: these are places where each compiler implementation can decide on the behavior but documents how the choice is made. These sorts of decisions are often left to the compiler in places where they would depend on underlying hardware and operating system architectures.
- 4) Undefined behavior: what happens when the program construct is used incorrectly or with incorrect inputs.

C and C++ Language Design


- C++ and C developers made design decisions:
 - Trust the programmer
 - Don't prevent the programmer from doing what needs to be done
 - Preventing the programmer from doing stupid things also prevents them from doing brilliant things
 - Don't prevent the code from talking directly to hardware
- Original use C and C++ were in secure environments
 - Isolated systems (eg. UNIX) that did not provide opportunities to hack
 - There was now wide-spread IT infrastructure
 - C and C++ were designed for specific tasks – security was not an issue

1-20

Every programming language designer makes decisions as to what they will allow the programmer to do in terms possible risk by allowing or prohibiting certain kinds of functionality. In the case of C++ , following the tradition of C, the decision was that it is up to the programmer to avoid risk. There are several reasons for this.

- 1) Very often C and C++ code interacts directly with hardware, especially in operating systems and embedded programming in devices, so that preventing certain kinds of operations like memory access might prevent the hardware from being used appropriately.
- 2) C and C++ code often has run in small environments (for example, limited device memory or real time responses) which means that the additional overhead of checking code at run time for illegal operations may produce unacceptable code overheads in terms of size and speed.

Bjarne Stroustrup once commented "*You can shoot yourself in the foot with C, but with C++ you can blow off your whole leg.*"



Module Topics


1. Introductions to Best Practices
2. Organizational Issues
3. C++ Standards
- 4. C++ Security Standards**
5. Implementation of Secure Coding

C++ Secure Programming Standard

- Developed by CERT to focus on secure programming
- One of the goals of the standard is:
 - "eliminating undefined behaviors that can lead to undefined program behaviors and exploitable vulnerabilities"
- Security flaws are not in the language itself
- Security flaws depend on how the language is used
 - The CERT standard focuses on how to avoid certain patterns of code use that can lead to security flaws

1-22

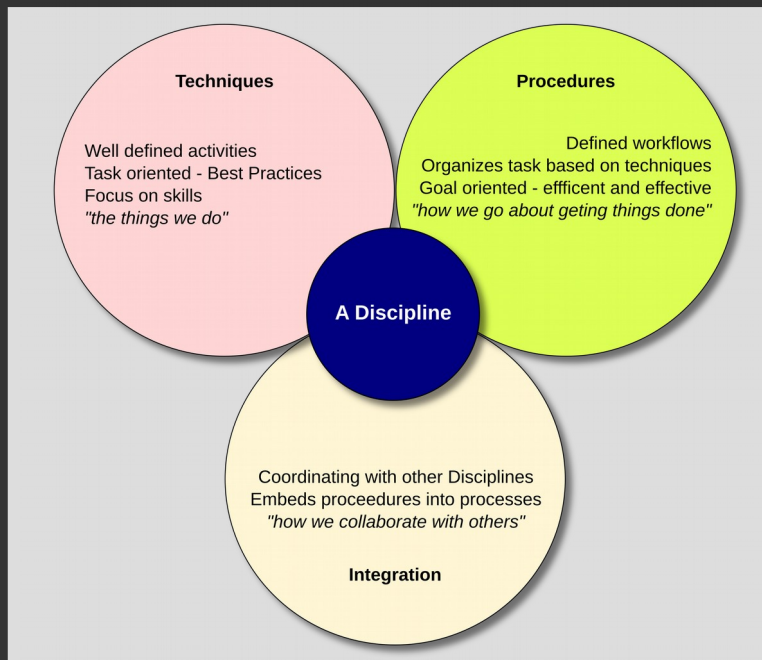
Security flaws are not at the language construct level but on the use of the language constructs in ways that are not correct from a program design point of view; for example deallocating memory and then writing to the deallocated address.



Module Topics

1. Introductions to Best Practices
2. Organizational Issues
3. C++ Standards
4. C++ Security Standards
- 5. Implementation of Secure Coding**

Secure Coding is a “Discipline”



1-24

This concept has been borrowed from Uncle Bob Martin's books on software craftsmanship.

To be successful, secure programming requires addressing a number of different aspects of code development. I have found over the years that introducing this concept and programming as a discipline, especially after giving examples out in the real world of other disciplines, it helps students conceptualize what the implementation of secure coding is all about.

You may want to come up with a few of your own examples of disciplines. For example, there are many examples in the armed services (doing things "The Navy Way"), running a restaurant kitchen, playing in a musical band.. and so on.

Discipline

- Defines techniques
 - Surgical techniques for a doctor / Cooking techniques for a chef
 - *Following the C++ Security Standards*
- Defines processes to accomplish tasks using techniques
 - Doctors use their tech to perform operations / Chefs use their techniques to cook meals
 - *Code development processes – e.g code reviews, collaboration*
- Defines how to integrate with other disciplines
 - Doctors need to coordinate with other operating room disciplines / Cooks need to coordinate with other kitchen and dining room staff
 - *Integration with QA, testing and design when writing secure code*

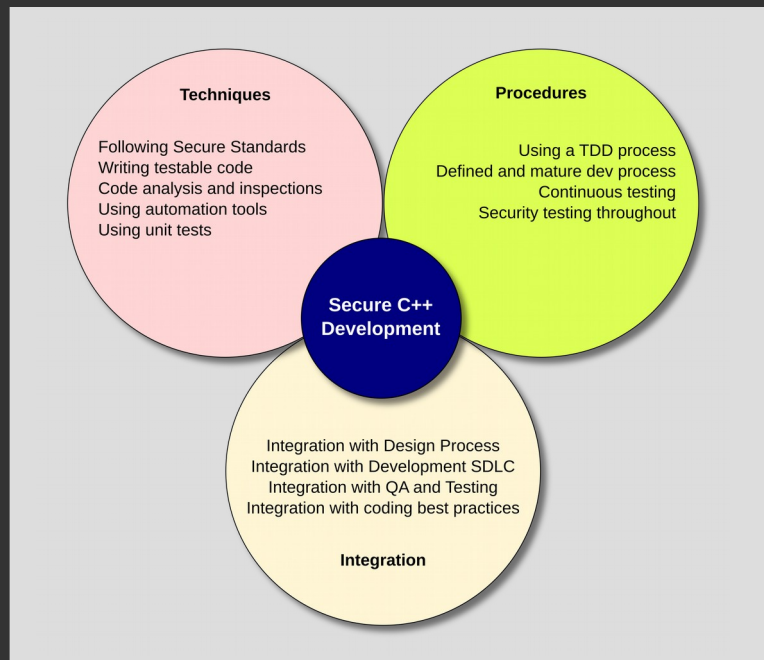
1-25

This slide assists in comparing Secure coding to the ways other professions work. Of course you should find your own example if you have one that you think works better for you.

Don't get hung up in class on the details of what a discipline is – it's just a teaching tool help students conceptualize what secure coding entails by pointing out analogies to similar things in other professions.

The main learning objective of this slide is that secure coding is made up of skills, procedures and way to collaborate; and in that sense it is just like many other professions. This secure coding discipline is not some unnatural theoretical construct but something solid and practical – that should be the take-away here.

Secure Coding is a “Discipline”



1-26

As Demming said

“It is not enough to do your best, you must know what to and THEN do your best.”

Sources of Insecure Code

- Malicious actor
 - Code that is intentionally insecure or unsafe
- Poor design
 - Application design is flawed, overly complex or incomplete
- Poor Coding
 - Code is not well written or has security flaws
- Insecure Programming Language
 - Language features allow insecure code or coding process allows for the introduction of insecure code

1-27

Malicious Actors: There are two basic types: hackers who attempt to breach the application from outside and saboteurs who compromise the actual source code that goes into the application. We are not going to deal with hackers in this course since the applications we are discussing are not intended to be exposed outside a private network. There are a number of ways to prevent malicious code from being injected into the code base which we will cover at a high level later in this module.

It is also important to keep in mind that even private networks can be the target of very sophisticated external malicious actors. The famous "Stuxnet" worm is an example where it was engineered to migrate from device to device on flash drives and other media looking for a target.

Poor Application Design: We won't be dealing with this topic in a macro sense but poorly designed applications often have significant vulnerabilities. For example, Google and other tech companies realized that while external access directly into their data was highly secure, there was no security on data travelling internally between computing centers which created a massive security hole. We will touch on application design only in so far as it relates to the design of C++ code.

Poor Coding: This will be the bulk of the course – the actual writing of secure C++ code.

Insecure Programming Language and Operational Environment: C and C++ are both programming languages that allow the programmer to do very powerful things, but their design model is that it is the programmer's responsibility not do things that would create code that is not secure or safe. Other programming languages have different design philosophies. For example, Java was designed to be more restrictive in terms of preventing unsafe code than C++. The modern Rust programming language is designed to make it impossible to compile certain unsafe code constructs. What this means for C++ programmers is that the main responsibility for writing secure code is the squarely on the programmer.

Protection Against Insecure Code

- Code Base Management
 - Configuration management policies and procedures
- Coding Methodology
 - Approach to designing code – development process
- Coding Practices and Standards
 - Adherence to best practices and security standards
- Quality Assurance
 - Thoroughness and robustness of testing and software QA

1-28

This slide identifies the location where insecurities occur rather than the actual root cause of the insecurity. The same root cause may occur because of how the code is managed or because of how the code is designed or because of a failure of code quality practices.

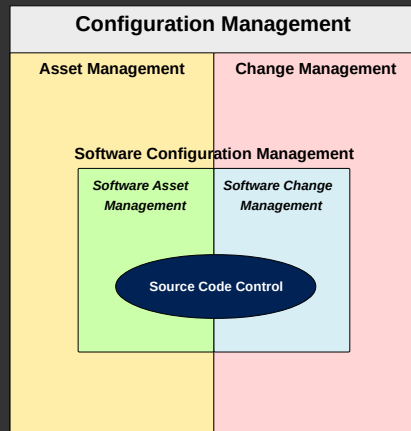
Code Base Management: Failure to secure the code base with a strong configuration management policy and accompanying procedures can open the door to both security and safety issues. Not only does strong CM remove the opportunity for a lot of malicious code insertion, it also allows for effective change and version management of the code so that vulnerabilities, once fixed, remain fixed.

Coding Methodology: What the last twenty years have taught us as a programming community is that there are specific ways to work at developing code that allow programmers to work faster, more productively and with fewer errors. One such methodology is test driven development, which we will touch on in a later section in this module.

Coding Styles and Practices: This is what the rest of the course after this module will be focused on.

Quality Assurance: The importance of code quality assurance cannot be understated. In addition to standard testing techniques, for secure coding we want to also have code reviews and walk throughs as well as engaging in third-party (if possible) exploratory testing.

Version Control Systems



Asset Identification:

Cataloging and identifying software assets, recording their properties, locations and where and how they are used.

Asset Control:

Policies for how an asset may be used or accessed.

Change Control:

Evaluating requested changes and evaluating the effects of a change.

Change Tracking and Auditing:

Identify where, when and why a change was made and by who.

1-29

Version control systems are critical for the secure coding. This is not a course on CM but it should be noted that one of the most important features of an SCM in a secure environment is to make it very difficult for bad actors to alter code without detection. Having to check code in and out, along with meta-data that tracks the coder, location and changes made and so on make it extraordinarily difficult for bad actor to inject malicious code into the code base.

Change control is also an important aspect of SCM since it requires an analysis of the impact of a code change before it is made and very often changes are rejected, at least in the form proposed, because of their possible impact of the safety and security of the overall application.

Secure SCM is Policy Based

- SCM tools do not "do" SCM
 - They only automate what you already do
- Effective SCM requires policies
 - The rules that govern who can to what to which item
- And procedures
 - Descriptions of the steps to be followed for policy compliance
- Meta-data analysis and reporting is critical
 - It is useless if it is not used

1-30

This cannot be emphasized enough, SCM tools only automate your SCM policies and procedures.

An SCM policy is a set of rules that defined roles, responsibilities, access rights and other privileges for users. It also defines under what conditions changes can be made. For example if it is a code change then it has to be approved by the change manager, or if it is new code, it has to pass a design review.

Procedures are generally the workflows that allow people to follow the policies while they execute their tasks. A single policy may have multiple procedures depending on the context of the work, while a single procedure may satisfy several policies.

Meta-data is invaluable only if it is used. In addition to flagging specific non-standard or suspect operations, it can also be used for forensic and historical auditing to either track down a possible bad actor or the root cause of some unsafe code and the reasons why it was written that way. However with modern analytic and business intelligence tools, creating statistical profiles of activities can often alert us to unexplained variances which might be the result of some form of intrusion or a breakdown in the coding process that could place the security of the code at risk – and ambiguous or incomplete specification for example.

Pair and Triad Programming

- Pair: two programmers work on the same code
 - One codes and the other reviews as the code is written
 - Roles are switched at intervals
- Triad: a third reviewer inspects the code
 - Takes place at milestones in the code development
 - Helps avoid "folie aux deux" group think
- Security enhancement
 - Programmers are paired randomly each day
 - Reviewer is assigned randomly
 - Prevents collusion

1-31

Every programmer has had the experience on struggling with a bug that just seems to be unsolvable, perhaps for hours on end, poring over every line of code with a fine tooth comb – and then a colleague walks but, takes a glance at your screen and points the bug out. Once it's pointed out, you wonder how in the world you didn't see it.

Pair programming takes that principle of "two sets of eyes are more effective than one" and turns into a "review as you code" process. We won't get into the details of pair programming in this course but it is an example of a programming methodology. Pair programming also has a security component to it as well since it makes it difficult for a single programmer to have the opportunity to inject malicious code while someone is reviewing everything they do. For improved performance and security, the programmers rotate their partners at regular intervals.

Triad programming is an extension of pair programming which has been implemented in several high-security coding environments. The concern was twofold. The first was that two programmers could start to develop something called a folie aux deux where they both start to think alike and can't pick up each other's mistakes. The second was that if two programmers were colluding to inject malicious code, all they had to do was wait until they were paired up, at which point the opportunity would present itself to make the malicious changes.

In triad programming, a third person is selected at random to review the code of the pair programmers at the end of each day in addition to the programmers being paired at random at the start of each day. This meant that multiple sets of eyes would be looking at the code on a regular basis which would effectively almost eliminate any possibility of either malicious injection of code or cases of folie aux deux.

Coding is Implementing Solutions

- We never start coding until we know exactly what the code is supposed to do.
 - *Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little.* Bertrand Meyer
 - *The most important single aspect of software development is to be clear about what you are trying to build.* Edsger Dijkstra
 - *First, solve the problem. Then, write the code.* Donald Knuth
 - *Conceptual integrity is the most important consideration in system design.* Fred Brooks

1-32

Well-written code comes implementing well-designed code. The quotes in the slide are all from some of the most brilliant computer scientists/engineers of the last 50 years. When we look at code from a security perspective, we find that code that is written without a clear design or in an ad hoc manner tends to be poorly structured, buggy and much more likely to be unsafe and insecure.

Part of the design process is the proactive analysis of where unsafe code could occur or vulnerabilities could exist that could be exploited by a bad actor.

Test Driven Development

- TDD is a popular best practice for programming
 - Allows programmers to write better code faster
 - Requires programmers follow best practices for coding
- In TDD, developers write unit tests first
 - All code is fully tested as new code is added
 - Forces solution to be developed before code is written
- Supports refactoring
 - Continuously improving code structure without altering functionality or introducing bugs

1-33

There is no TDD standard or canonical TDD process. Like a language, TDD has different "dialects" but they all tend to share a more or less common core of ideas. This is because TDD is often modified and adapted to fit into specific environments or different methodologies. This has resulted in many developers using "lazy" TDD where TDD practices are used in a very casual fashion or are inconsistently applied. Lazy TDD runs into problems when the projects scale up and the lack of discipline makes it difficult to manage the the more increased complexity.

There have been a number of research studies to see if TDD produces benefits. On average code produced with TDD has a lower defect density and has better coverage. However, the studies shows mixed results as to increases in programmer productivity and decreased development time.

The studies also show that often there is an initial increase in development time but off-set by "down the road" savings in development and application support. One of the more interesting effects is that the resulting application components are more often loosely coupled which suggests a positive impact of application design. As well, code produced under TDD scores higher client satisfaction measures on average.

Some studies show dramatic improvements in programmer productivity, but some show no change, and a few show a decrease. When the TDD discipline is followed rigorously, then we do see lower defect density, reduction in rework, productivity increases and reduction in project times. However, when lazy TDD is followed or when TDD is implemented inconsistently or only partially, productivity is impacted negatively.

[One factor that explains the mixed results is how rigorously TDD is followed because the best results from using TDD come from engineering and industrial environments where TDD is applied rigorously and consistently because the organizational culture is focused on quality of code, risk and productivity. Neutral or negatives results tend to come from academic or non-industrial contexts where the culture is not highly quality oriented and TDD tends to be only partly applied or inconsistently applied.

TDD emphasizes working through the tests first, then writing the code which forces programmers to immerse themselves in the problem before writing any code. This also forces a critical analysis as to whether the developers know what the software they are building is supposed to do.

If you can't figure out what the right result of a test should be, then you don't know what your code should do in that situation.

This seems to force the programmer to take a global view problem to be solved before crafting any details of a solution. However, and perhaps even more importantly, TDD is consistent with the cognitive mechanisms used to solve problems which are now starting to be formalized in design process like the Stanford Design Process.

I've worked on three different development teams using TDD and on several more teams that didn't. I can tell you from first-hand experience that TDD produces code that has orders of magnitude fewer unit-level bugs, far fewer functional bugs, and an exponentially higher probability of meeting stakeholder expectations when compared to code produced by conventional programming techniques



Lisa Crispin

1-34

Crispin may not be a familiar name to many but she is one of the leading figures in the Agile testing community – one of the gurus so to speak.

However, more importantly, her comments here echo what many have reported. We are not going to try and wage a battle to prove the case here, but rather provide the suggestion that the evidence is there to support the case for TDD.

Some Empirical Evidence

- A number of research studies evaluated TDD benefits
- On average code produced with TDD:
 - Has a lower defect density
 - Has better coverage
 - Shows mixed results as to increases in programmer productivity and decreased development time
 - Is often more loosely coupled
 - Has higher client satisfaction measures on average
 - Is often less complex and easier to maintain

1-35

Again, not to try and prove the case, just raise some interesting points

By better coverage, we mean that there are fewer cases that “fall through the cracks” and the code is essentially more robust.

Loosely coupled is a real positive. However, it is not TDD itself that makes the code loosely coupled but rather the fact that it forces developers to follow better design practices that in turn result in more loosely coupled code.

This slide is looking at the benefit of TDD on the code

Some Empirical Evidence

- A number of research studies evaluated TDD benefits
- Overall benefits
 - Some studies show dramatic improvements in programmer productivity, some show no change, and a few show a decrease
 - Often shows initial increase in development time but off-set by “down the road” savings in development and application support

1-36

This slide looks at the benefit of TDD to the overall process.

Notice that the benefits to the overall development process are not as clear cut as the benefits to code. However, the next few slides discuss the possible reasons for this.

A Critical Success Factor

- Explaining the mixed results
- The best results for TDD come from engineering and industrial environments where:
 - TDD is applied rigorously and consistently
 - Culture is focused on quality of code, risk and productivity
- Neutral or negatives results tend to come from academic or non-industrial contexts where:
 - Culture is not usually highly quality oriented
 - TDD tends to be only partly or inconsistently applied
- Results correlate with process maturity

1-37

The mixed results of TDD depend on how “process mature” a team is in following TDD. In other words, if the team applies lazy TDD or only applies TDD some of the time, then they don’t get the full benefit.

TDD is like a physical training program. If you are training to run a marathon, you don’t get too far if you only implement part of a training program, or only train once in a while. Or like a diet where you don’t get results if you only follow part of the diet or only practice for a part of each day.

TDD is like any process. If you aren’t familiar with the CMM process maturity levels, this is a perfect example of where you need to be committed to the process to master it, then you really start to see benefits.

There is a section in the manual on the CMM process maturity that you can refer to as background.

Programmer Comments on TDD

"I'm writing code faster with less rework and less profanity"

"It's like the code is writing itself, I'm not doodling in code any more trying to figure out why I wrote that piece of code a month ago"

"My code is simpler, cleaner and easier to understand"

"Amount of rework is reduced since bugs are caught early, which means I go home on time"

"Continuous regression testing means that adding new code doesn't break my existing code"

"Programming is fun again. I don't spend all my time on bug hunts and I get to think instead about optimizing my design and implementation"

1-38

These are actual comments collected by the author from either conversations with professional programmers who adopted TDD or students who applied TDD after going through the authors TDD course.

Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying a series of small behavior-preserving transformations, each of which "too small to be worth doing".

However the cumulative effect of each of these transformations is quite significant. By doing them in small steps you reduce the risk of introducing errors. You also avoid having the system broken while you are carrying out the restructuring - which allows you to gradually refactor a system over an extended period of time.



Martin Fowler

Refactoring

- Refactoring:
 - Changes the structure of existing code
 - Does not changes the the functionality of the code
 - Uses TDD to ensure code changes do not cause bugs
- Refactoring is usually done:
 - After the functionality of the code changes
 - As part of a design change
 - After a bug has been fixed
 - After a code review

1-40

Programmers have always made code changes, but the classic problem is that when we make a change, we break the code.

Refactoring provides a procedure for making code changes without breaking code. Again, this is a great opportunity to poll the class about their experiences with making changes to the structure of the code which winds up introducing bugs.

Well Structured Code

- Code that is well designed and well structured:
 - Is easier for programmers to understand
 - Is easier to modify
 - Is easier to maintain
 - Is more elegant and efficient
- When unplanned changes to code take place:
 - The structure of the code degrades
 - The degradation to the code is cumulative
 - Eventually the code devolves into an unstructured mess
 - Unless refactored, changes to code increase code “entropy”

1-41

A follow up from the previous slide. Most developers have a good sense that code that is well designed is simpler and easier to use. Code entropy always increases unless the code is regularly refactored.

Code Smells

- A code smell is:
 - “... a surface indication that usually corresponds to a deeper problem in the system” according to Fowler
 - Where the code does not support good design practices
 - An indication a refactoring needs to be done
- This is closely related to the idea of an anti-pattern
 - Anti-patterns are recurring patterns of how developers go from a problem statement to a bad solution
 - Anti-patterns occur everywhere: project management, architecture, etc.
 - Code smells are sometimes referred to as code anti-patterns
 - Refactorings also exist for anti-patterns

1-42

We will deal with these later in the course

A good question to get students thinking about code smells is to ask them if they can think of a time when they looked at some code and, even though the code worked, it just “didn’t look right”

That would be a code smell.

More than the act of testing, the act of designing tests is one of the best bug preventers known.

The thinking that must be done to create a useful test can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest.

**If you can't test it, don't build it.
If you don't test it, rip it out.**

Boris Beizer



This is one of my favourite quotes since it seems to be something I constantly have seen verified in practice, although ironically it comes from the 1982.

- 1) Have every found that thinking through the things that could go wrong before you do something makes the job go smoother?
- 2) Why do you think this would be true? Why would designing tests eliminate bugs?

Quality Assurance

- Regular QA consists of
 - Static code reviews by review teams
 - Code walk throughs of potentially vulnerable code
 - Automated code analysis
- Coding standards
 - Should be reviewed and updated regularly
 - Should focus on preventative coding
 - Should incorporate secure coding standards
- Coverage Analysis
 - Ensures there is no "unaccounted for" code

1-44

We will deal with these later in the course

A good question to get students thinking about code smells is to ask them if they can think of a time when they looked at some code and, even though the code worked, it just “didn’t look right”

That would be a code smell.

General Best Practices

- Validate input
- Heed compiler warnings
- Architect and design for security policies
- Keep it simple
- Default deny
- Adhere to the principle of least privilege

1-45

1. **Validate input.** Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user-controlled files.
2. **Heed compiler warnings.** Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code. Use static and dynamic analysis tools to detect and eliminate additional security flaws.
3. **Architect and design for security policies.** Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.
4. **Keep it simple.** Keep the design as simple and small as possible. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex.
5. **Default deny.** Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted.
6. **Adhere to the principle of least privilege.** Every process should execute with the the least set of privileges necessary to complete the job. Any elevated permission should only be accessed for the least amount of time required to complete the privileged task. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges.

General Best Practices

- Sanitize data sent to other systems
- Practice defense in depth
- Use effective quality assurance techniques
- Adopt a secure coding standard
- Define security requirements
- Model threats

1-46

7. **Sanitize data sent to other systems.** Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem.


8. **Practice defense in depth.** Manage risk with multiple defensive strategies so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment.

9. **Use effective quality assurance techniques.** Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions.

10. **Adopt a secure coding standard.** Develop and/or apply a secure coding standard for your target development language and platform.

11. **Define security requirements.** Identify and document security requirements early in the development life cycle and make sure that subsequent development artifacts are evaluated for compliance with those requirements. When security requirements are not defined, the security of the resulting system cannot be effectively evaluated.

12. **Model threats.** Use threat modeling to anticipate the threats to which the software will be subjected. Threat modeling involves identifying key assets, decomposing the application, identifying and categorizing the threats to each asset or component, rating the threats based on a risk ranking, and then developing threat mitigation strategies that are implemented in designs, code, and test cases.

A top-down view of a wooden desk. In the top right corner, a portion of a silver laptop is visible, showing keys like 'Q', 'W', 'E', 'A', 'S', 'Z', 'X', 'C', 'V', 'B', 'N', 'M', 'control', and 'option'. Below the laptop, a pair of black-rimmed glasses lies horizontally. To the right of the glasses is a white ceramic cup filled with dark coffee, with a yellow handle. In the top center, a small green succulent in a dark pot sits on the desk. A dark grey rectangular box with the text 'Module End' is positioned on the left side of the desk. The bottom right corner of the image shows a dark surface, possibly a tablet or another part of the laptop, with the text '1-47' in the bottom right corner.

Module End