Overview

CS205, System Programming and Architecture, was designed for students pursuing the OSU/PSU/UO variant of the CS Major Transfer Map (MTM). Oregon State University and Portland State University both require a lower division course focusing on the interface between high-level code and hardware (CS271 and CS201 respectively. Including the credits for these courses was a critical part of making the MTM workable. This course, while not exactly equivalent to their courses, articulates to those courses. CS205 is optional but strongly encouraged for students bound for UofO where, although it does not directly articulate to a single course, it does provide coverage of some material that is contained in their lower division sequence.

The core theme of this course is "What really happens when software runs?" Students should learn how fundamental parts of C programs map to assembly code and binary representations, how this assembly is determined by the Instruction Set Architecture of a machine, the high-level structures of a processor, and the basic facilities provided by an operating system. The particular architecture and assembly studied is not proscribed - students exposed to any modern architecture should be adequately prepared for later courses.

For colleges that do not teach C as part of their CS161/162/260 sequence, this course also serves the essential role of introducing C programming so students are prepared for Junior year courses that assume a working knowledge of C. Thus, it is assumed that time will be devoted to learning the basics of C programming in addition to learning how C gets mapped to assembly code. After this course, students should be able to write moderately complex, well-structured programs using C. Students should also have experience using a debugger on their code.

Capture The Flag (CTF) style activities that require reading assembly in context and basic dynamic analysis of code using a debugger are a key component of the course. These are intended as a platform to develop hands on skills examining compiled code in static and dynamic contexts and are assumed to be an ongoing thread in the course as opposed to a standalone unit. These activities, or equivalent ones, must be a part of any implementation of CS205 in order to be acceptable for transfer. To aid instructors in meeting this requirement, PSU is providing access to the CTF system they have developed (see CTF section for details).

The 10th week of this course is designated as time to apply and expand the ideas of the course in a particular context. Three possible approaches - ones that focus on cybersecurity, operating systems, and designing for performance - are briefly outlined in this document to provide suggested uses for this week.

Course Outcomes

Any implementation of CS205 should include the following based outcomes.

1. Describe the major components of computer architecture; explain their purposes and interactions and the instruction execution cycle.
2. Describe a basic instruction set architecture, including the arithmetic, logic, and control instructions; user and control registers; and addressing modes.
3. Do simple arithmetic in hexadecimal, decimal, and binary notation, and convert among these notations.
4. Explain how data types such as integers, characters, pointers, and floating point numbers are represented and used at the assembly level.
5. Write C language programs that use control structures, functions, IO, arrays, and dynamic memory.
6. Describe each step of the compilation process by which C language programs are transformed into machine code.
7. Explain how high-level programming constructs such as arrays, structures, loops, and stack-based function calls are implemented in machine code. Recognize and reverse engineer same.
8. Demonstrate and use a debugger to analyze program flow, inspect register and stack contents.
9. Identify and fix performance issues in C programs that are caused by machine level concepts.
10. Explain how the information in this course is important within the overall context of computer science.

CTF Activities

As mentioned above, implementations of CS205 are expected to use Capture The Flag style activities to develop hands on skills working with machine level code. Although Capture The Flag style activities derive from the cybersecurity community, the goal for these activities in this course is not the development of specific cybersecurity skills. Instead, their goal is to develop the skills necessary to inspect running machine-level code and the state of memory and understand what is going on. Those are the skills that serve as a foundation for future courses in operating systems, cybersecurity, compilers, embeded systems, and any other upper division work involving low level code.

Similarly, while CTF originated as part of competitions where teams compete to solve a series of increasingly fiendish challenges, implementors of this course are explicitly encouraged NOT to utilize a competitive structure or focus on open-ended puzzles. Activities adopted from the CTF community should be modified to focus on pedagogical effectiveness instead of their utility for differentiating competing teams based on problem solving skills.

Specific skills students are expected to develop are:

1. Use basic tools (e.g. objdump) to inspect compiled code.
2. Read assembly in context and understand it. In particular, understand code generated by a compiler.
3. Use a debugger (command line or visual) to inspect running code and memory.

Sample activities:

1. Given a binary, use basic static analysis (objdump) to find a password or other information.
2. Given a binary, use basic static analysis (objdump) to figure out the proper input to give to cause a desired output.
3. Given a binary, use a debugger to inspect it while running to figure out the proper input to cause a desired output.
4. Modify an executable with a hex editor (or running code with a debugger) to enable/disable functionality (change a branch target).
5. Examine a program and engineer a basic buffer overflow on a stack based buffer to modify other local data or the return address. Or use a buffer to leak information about a running program.

More advanced CTF-style challenges likely go beyond the intended requirements of this course. This would include problems that involve: obfuscated code; detailed knowledge of OS constructs (user accounts); networking; buffer overflows that introduce new code; return oriented programming; or heap based exploits.

PSU CTF Activities

Dr. Wu-chang Feng at PSU developed a web-based system, https://oregonctf.org/, for running CTF challenges and CTF style learning activities. A series of activities has been developed at PSU to support teaching their CS201. Each of these activity requires students to exercise skills to retrieve a password that is unique to them that verifies they completed the activity. So while students can help each with the techniques needed to solve each challenge, they cannot provide each other with answers.

Dr. Feng and PSU have agreed to make the system available for schools implementing a CS205 to use. Schools can either:

- Host their own server using this docker image: https://hub.docker.com/repository/docker/wuchangfeng/metactf.
- Ask PSU to host a server as a subdomain at oregonctf.org. Contact wuchang@pdx.edu to set this up.

If you would like to test out the site, you can use https://cs201.oregonctf.org with the demo account credentials listed on the login page.

The topic outline below lists relevant CTF activities from the PSU problem set under each section.

CS205 Topic List

The listed ordering of topics should in no way be considered prescriptive. While the topics listed below generally progress from fundamental to more advanced, any particular implementation of CS205 will likely find it advantageous to rearrange or crosscut topics. In particular, because this course covers both C programming and how C code runs on hardware, there are two possible approaches to teaching it: 1) focus on C programming before diving into assembly and how the code runs; 2) iterate through areas of C programming and how they are implemented. The topic list is structured in the later manner: it describes the expected boundaries for C and lower levels of each topic simultaneously. That should not be understood as precluding teaching the course by first focusing on C and then covering assembly/machine level topics.

1 Introduction to Systems

The fundamentals of computer systems and architecture that are critical for understanding assembly and high level performance concerns that are affected by architecture.

1. Fundamental components of a computer system and the fetch, decode, execute cycle.

2. Memory hierarchy and role of caches. Detailed study of caches not required. Focus should be relative performance of different levels of the memory system and basic principles of cache management: block transfer and temporal and spatial locality.
3. Role of Operating System as hardware manager.

2 C Development Fundamentals

Writing and debugging a C program that does basic IO. Other topics in C programming are broken out with their assembly counterparts but could be taught with the fundamentals to make a larger initial module on C programming.

1. Role of C as portable assembly and high level language.
2. Structure of a basic C program and use of basic IO and file instructions (fopen, printf, fgets, etc...).
3. Compiling and running C code from the command line.
4. Use of a debugger to examine running code at both the C and ASM level. Students should be familiar with setting breakpoints and examining the contents of registers and memory associated with symbols.

Relevant PSU CTF(s)

Ch3_00_GdbIntro, Ch3_00_GdbPX, Ch3_00_GdbRegs

3 Compiling, Linking, and Loading

How C code is turned into object files and executables and the structure of those files. Across all of these areas, the focus is building the level of understanding important for a developer who will use C in future courses and who will be examining compiled code to understand it. Precise implementation details of any particular object format, linking process, etc... should not be an emphasis.

1. Compilation process - roles of: preprocessor, C compiler, assembler, linker. Individual components can be treated as black boxes, but students should recognize the kinds of errors messages that can result at each phase (linking vs compilation errors).
2. Basic structure of object files. Focus should be on understanding different segments and their roles.
3. Symbols, symbol tables, and symbol resolution. Emphasis should be on identifying and how symbols will be matched across object files/compilation units.

4. Relocation and position independent code. Again, focus should not be the details of any particular implementation, but on the kinds of transformations that happen as code is linked and loaded.
5. Static vs dynamic linking.
6. Use of basic static inspection tools (e.g. objdump)

Relevant PSU CTF(s)

Ch1_LTrace, Ch1_Readelf, Ch7_13_LdPreloadGetUID

4 Data Representation

How data is represented at a low level and how that shapes C programming. Understanding that these are determined by the architecture of the machine is critical - knowing the details of any particular architecture is not.

1. Binary, Decimal, and Hexadecimal representations. Converting from one form to another.
2. Data sizes. Bytes and words as the fundamental unit sizes.
3. ASCII char representation.
4. C integer data types and their sizes. Emphasis on platform dependence of these.
5. Enumerations.
6. Machine instruction representation. How features like opcode, registers, and memory addresses are stored in an instruction.
7. Endianess.
8. Casting in C.

Relevant PSU CTF(s)

Ch2_01_Endian, Ch2_01_Showkey, Ch2_03_IntOverflow, Ch3_02_AsciiInstr

5 Signed Arithmetic and Floating Point

How we do integer math in C and ASM and floating point math in C. (Coverage of floating point instruction in ASM is not required.) How negative and fractional values are represented at a binary level.

1. Signed integer representation in 2s complement. Negation of signed values at the bit level. Recognition of important patterns (0xF...F is -1; a leading 1 is a negative number).
2. Signed vs unsigned arithmetic. Overflow and its detection. When to use which format.
3. Floating point representation. Memorization of a particular format is not required, but students should be exposed to an IEEE or IEEE-like format and understand the basic features of floating point representation: how representational resolution depends on magnitude, the existence of minimal/maximal representable values, and nan/infinity.
4. The floating point types and their use in C.

Relevant PSU CTF(s)

Ch2_03_TwosComplement, Ch2_05_FloatConvert

6 Bitwise Operations

How to manipulate bits in ASM and C.

1. Bitwise logical operations and their use to manipulate individual bits.
2. Shift/rotate operations.
3. Use of shifts to multiply/divide binary values. The difference between arithmetic and logic shifts.

Relevant PSU CTF(s)

Ch2_03_XorInt, Ch3_05_LorStr

7 Control Structures

C level control structures and their implementation at the ASM level.

1. Conditional structures (including switch) and loops in C.
2. ASM comparison instructions and condition codes.
3. ASM jump/branch instructions including conditional branches.
4. Jump/branch tables.
5. Implementation of C control structures in ASM.

Relevant PSU CTF(s)

Ch3_06_SwitchTable

8 Memory and Pointers

How memory is accessed. Of critical importance is understanding the role of registers in the implementation of complex C expressions (or in all expressions for load/store architectures).

1. Pointers, addresses and dereferencing in C.
2. ASM movement instructions using immediate values, registers, and memory.
3. ASM addressing modes.
4. The use of the stack to store data via ASM push/pop operations. How that stack is layed out in memory and how the stack pointer is used to maintain it.

Relevant PSU CTF(s)

Ch3_04_FnPointer, Ch3_04_LinkedList

9 Dynamic Memory

Allocation/deallocation of memory in C and the standard memory model. Memory management in ASM would be an advanced week 10 extension.

1. Use of free/malloc in C.
2. Heap vs stack. Strengths/weaknesses of allocations in each region.
3. The "standard" process memory model (and how reality often is more complicated).

10 Functions

Functions in C and their implementation in assembly.

1. Functions in C.
2. Pass by value vs reference.
3. ASM level calling convention - how values are passed and returned.
4. ASM implementation of local memory. Frame pointer and stack allocation/deallocation.
5. Recursive functions - how the stack mechanism allows for reentrant code.

Relevant PSU CTF(s)

Ch3_00_GdbRegs, Ch3_07_ParamsRegs, Ch3_07_ParamsStack, Ch3_07_SegvBacktrace, Ch3_07_StaticStrcmp

11 Arrays and C-strings

Use of arrays and C-strings and working with arrays in ASM.

1. Array allocation (fixed and variable length) and use in C.
2. Multidimensional arrays in C.
3. Pointer arithmetic and its relation to array indexing.
4. Accessing individual array elements and looping through arrays in ASM.
5. C-string related conventions and functions: NULL termination and basic string library functions.
6. Stack based buffer overflow attacks.

Relevant PSU CTF(s)

Ch3_07_CanaryBypass, Ch3_07_StackSmash, Ch3_08_Matrix

12 Heterogeneous Structures

Fundamental tools for grouping (possibly) heterogeneous data.

1. Structs.
2. Unions.
3. Data alignment and byte packing/padding in structures.

13 Optimizations in C

Basic optimizations that compilers do related to concepts from this course. Exploring the effect of level optimizations in C code. (strength reductions, reducing procedure calls & memory references, etc...). Coverage of this topic is not expected to be exhaustive - the goals are awareness of why compiled code often looks very different than what one would expect and how the kinds of low level efficiencies.

1. The kinds of optimizations that compilers routinely make (function inlining, dead store elimination) when compiling code.
2. The effect of array stride order on performance.
3. Relative performance of floating point and integer math - why we generally favor ints unless we need floating points.
4. Basic performance optimizations available to high level language programmers: avoiding loop inefficiencies, reducing procedure calls and memory references, strength reductions.

Relevant PSU CTF(s)

Ch5_08_LoopUnroll

Sample Schedule

Wk	Topic
1	Introduction to Systems C Fundamentals
2	The Compilation Process Object files Data Representation
3	Data Representation Continued Arithmetic Operations Floating Point
4	Bitwise Operations Control Structures
5	Memory and Pointers Dynamic Memory Management
6	C Functions Implementation of Functions
7	Arrays and C-Strings
8	Structs, Unions, Alignment
9	Performance & Optimizations in C
10	CS205 Topics in Context

Week 10 Options

The tenth week of the course is designed to provide time for instructors to review critical material, finish up CTF work, and to expose students to applications of concepts from the course in the context of more advanced topics. While instructors are free to devise their own approach to doing so, three application areas are provided below as suggestions. For each, some resources are provided as a starting point.

1 Cybersecurity

Cybersecurity related knowledge and skills are undergoing a rapid increase in interest among students and employers. Many code level security topics depend directly on the knowledge and skills in this course. A deeper dive into cybersecurity related topics provides an opportunity to emphasize the practical CTF challenges and tease more advanced ideas like return oriented programming.

1.1 Reverse engineering

Reverse engineering of software is often done in a static fashion – take a binary and pop it into a disassembler (ghidra, Cutter (r2 or rizin), IDA). This typically results in a listing of assembly code, either in list or call-graph form (sometimes both). Being able to read ASM, in any form, makes being able to reverse engineer software that much easier.

A good in class exercise would be to take a program you know the structure of, and demonstrate how to reverse it using Cutter. Even better, take a lab that involved dynamic analysis (gdb bomb or similar) and redo the exercise statically.

Having a sample binary to pop into Cutter here would be a good short HW – have them write a description of what it does.

1.2 Firmware hacking

Most IoT devices aren't running x86 processors. Typically, they run some ARM variant, MIPS, or PPC. While this class doesn't cover those other architectures, knowledge of ASM is cross-functional. In other words, knowing ANY assembly language makes it easier to learn assembly language.

A fun alternative to the above suggested homework is to do the same thing. . . but with a binary from a MIPS or PPC system.

1.3 ROP

Exploit development nearly always has an assembly component. Even a simple buffer overflow requires some of the text being used to overflow the buffer to be written in ASM. This is known as shell code. One common approach to getting the shell code to run is to use a technique known as return oriented programming, or ROP. At a high level, ROP involves leveraging `ret`-like instructions to eventually cause a jump to an address you can control.

A demonstration of a simple ROP chain would be a good in-class exercise, as well as talking about tools such as ropper or other similar gadget generation tools.

TODO - Kevin to provide a short outline.

2 Operating Systems Programming

Students will generally go on to take an Operating Systems course that is very dependent on the skills developed in this course. Week 10 of CS205 is an opportunity to provide students with a consumer perspective on many of the topics they will study in depth in that OS course.

2.1 Exception Handling

1. Exception number
2. Exception table, aka, exception dispatch (jump) table, etc. ISA specific terminology.
3. Exception handler. No discussion of up and bottom halves, just the basic concept.

2.2 System Calls – OS As Privileged Service Provider

System calls are synchronous (traps). At the very least, there should be a detailed example of a simple C program making one or more system calls mapped down to assembly. The following exemplar is meant to be walked through line-by-line. This will help reinforce that system calls look just like function or library calls to the programmer. Modes will be introduced in the process discussion, below. The emphasis should be on POSIX-compliant system calls.

1. Processes and logical flow control.
2. Concurrent Flows - basically about a gut feel for what it means for 2 processes to be concurrent. This is a natural follow-on to exceptions and allows us to introduce the idea of a context switch.
3. Process Address Space Layout

4. User and Kernel Modes
5. The Context Switch
6. System Call Error Handling
7. Process Control - fork/exec/wait

Relevant PSU CTF(s)

Ch8_05_PsSignals, Ch8_05_Signals

Resources:

1. Computer Systems - A Programmers's Perspective, Ch 8; Bryant and O'Hallaron

3 Data Oriented Design and Optimization in Games

While data oriented design and other optimizations are useful in any programming where performance is a high priority, for many computer users, video games are the most computationally expensive software packages run on a routine basis. They are a high interest place to explore the kinds of considerations that go into designing performant software. Topics including memory access in arrays and compiler optimizations can be reviewed and expanded on and more advanced topics like branch prediction can be teased.

Resources:

1. The book Game Programming Patterns - available for free online: https://gameprogrammingpatterns.com/. Chapter 17 is likely the most relevant. Chapters 18-19 and 11 also of interest. Examples in the book are in basic C++.
2. Game Engine Architecture, Jason Gregory. Chapter 3 is most relevant. https://www.gameenginebook.com/toc.html
3. Seminal 2009 presentation by Tony Albrecht on optimizing for the PS3: http://harmful.cat-v.org/software/OO_programming/_pdf/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf

   A 2017 followup talk by Tony Albrecht: https://www.youtube.com/watch?v=VAT9E-M-PoE
4. Data Oriented Design in game development talk by Mike Acton: https://www.youtube.com/watch?v=rX0ItVEVjHc

5. Data Oriented Design to make a faster HTML renderer: https://www.youtube.com/watch?v=yy8jQgmhbAU
6. Matt Godbolt (of godbolt.org fame) on the cool magic that compilers do and why you should not assume you can out clever them with hand optimizations: https://www.youtube.com/watch?v=w0sz5WbS5AM
7. Struct of Array vs Array of Struct organization: https://medium.com/@savas/nomad-game-engine-part-4-3-aos-vs-soa-storage-5bec879aa38c