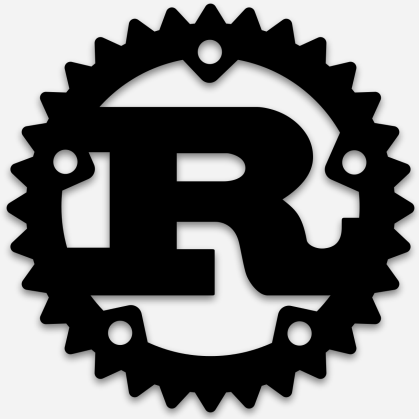
Secure Programming

Mustakimur R. Khandaker



Lecture and Office Hours

Instructor: Mustakimur Rahman Khandaker.
Office: 547 Boyd Graduate Studies Research Center.
Email: mrkhandaker@uga.edu

Office Hours: N/A.

Lecture time and location:
N/A

Monday	N/A
Tuesday/Thursday	N/A

Instruction Policy:

- Course instructions will be (hopefully) in person.
  - Instruction policy depends on the UGA plan.
- Monday class will be mostly used for presentations and quizzes.
  - The quiz date will be declared a week before.
  - Will showcase lab assignments (coding and analysis).
  - Also, general discussion on project assignments.

Project and Presentation

Evaluation and Grading Policy:

- Quizzes: 10%
  - 5 quizzes.
  - held on eLC on Monday class.
- Program Analysis: 10% (AFL Fuzzing)
  - Students will assign open-source projects to run AFL fuzzer.
- Individual Assignments: 30% (Rust)
  - 5 assignments..
- Group Project: 30% (Rust) [3 members team]
  - The project should be maintained in GitHub.
- Presentation: 20% (2 presentations).
  - From a shortlist of top-quality research.
  - shared by 2 students and presented together.

Group Project Policy:

- A shortlist of good projects will be published.
- Details of expectations will be included in the handout.
- The project final deadline will be the last day of the course lecture.
  - There will be 3 intermediate release deadlines.
- A late release will receive a 2% penalty for each day.

Assignment Policy:

- A report including required data should be submitted for the program analysis homework.
  - Submit an archive on the eLC course page.
- The assignments will due on at Sunday 11:59 PM on the specific days.
  - Submit code on the eLC course page.
- Late submission will receive a 3% penalty for each day (maximum 3 days allowed).

Topics

Section I: Software Security.

I. Software vulnerabilities.
   A. Memory corruption.
   B. Synchronization bugs.
   C. Exploitation.
II. Software Analysis.
   A. Program analysis.
   B. Sanitizers.
   C. Fuzzing.

Section II: Memory-safe Language.

I. Basics.
   A. Cargo.
   B. Data types.
   C. Control-flow.
II. Memory safety.
   A. Ownership model.
   B. Lifetimes.
   C. Error handling.
   D. Smart pointers.
III. Complex types.
   A. Structs, enums, collectors.
   B. Generics.
   C. Traits.
IV. Thread safety.
   A. Concurrency.
   B. Shared memory.
   C. Channel.

Section III: Advanced Language Features.

I. Functional programming.
   A. Iterators.
   B. Closures.
   C. Pattern matching.
II. Engineering.
   A. Modules and privacy.
   B. Nonblocking I/O.
   C. Automated testing.
III. Rust special:
   A. Unsafe Rust.
   B. Macros.
   C. Crates.
IV. Applications.
   A. Operating system.
   B. Web framework.
   C. Blockchain.

Letter Grade

Letter Grade	Percentage
A	91 - 100%
A-	86 - 90%
B+	81 - 85%
B	76 - 80%
B-	71 - 75%
C+	66 - 70%
C	61 - 65%
C-	56 - 60%
D	51 - 55%
F	0 - 50%

System Programming

System programming involves the development of the individual pieces of software that allow the entire system to function as a single unit.

- It involves many layers such as the operating system (OS), firmware, and the development environment.
- It requires a great degree of hardware awareness.
- Its goal is to achieve efficient use of available resources, either because the software itself is performance critical or because even small efficiency improvements directly transform into significant savings of time or money.

C/C++ (The Good Parts)

Efficient code especially in resource-constrained environments.

Direct control over hardware.

Performance over safety.

- Memory managed manually.
- No periodic garbage collection.
- Desirable for advanced programmers.

C/C++ (And The Bad Parts)

Type errors easy to make.

- Integer promotion/coercion errors.
- Unsigned vs. signed errors.
- Integer casting errors.

Memory errors easy to make.

- Null pointer dereferences.
- Buffer overflows, out-of-bound access (no array-bounds checking).
- Format string errors.
- Dynamic memory errors:
  - Memory leaks.
  - Use-after-free (dangling pointer).
  - Double free.

Cause software crashes and security vulnerabilities.

Example: C/C++ is Good

Lightweight, low-level control of memory

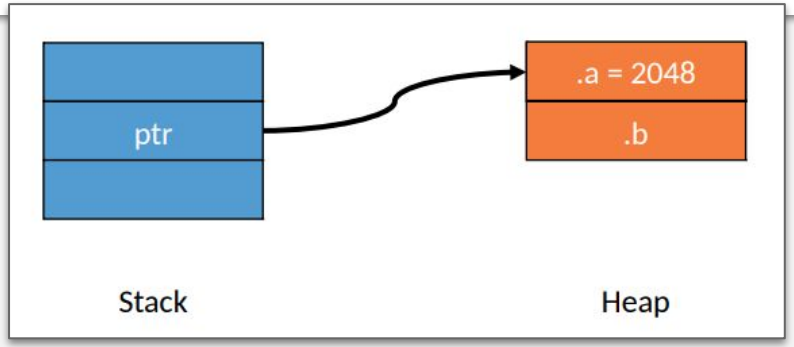
```
typedef struct Dummy { int a; int b; } Dummy;

void foo(void) {
    Dummy *ptr = (Dummy *) malloc(sizeof(struct Dummy));
    ptr->a = 2048;
    free(ptr);
}
```

Precise memory layout

Lightweight reference

Destruction



Example: C/C++ is not Good

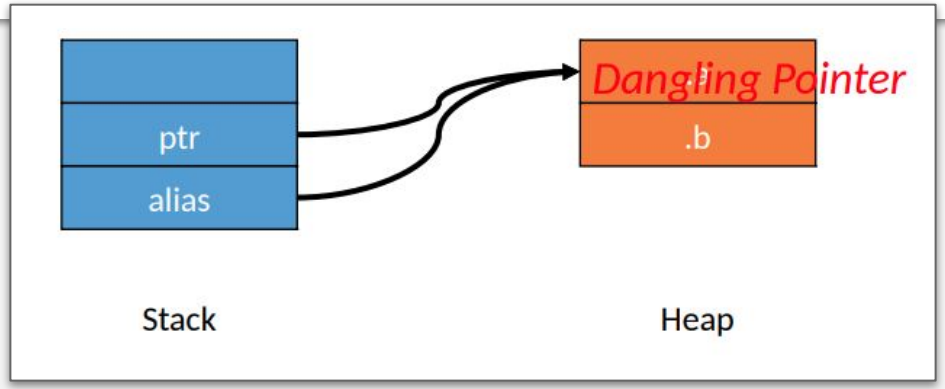
```
typedef struct Dummy { int a; int b; } Dummy;

void foo(void) {
    Dummy *ptr = (Dummy *) malloc(sizeof(struct Dummy));
    Dummy *alias = ptr;
    free(ptr);
    int a = alias.a;
    free(alias);
}
```

Use after free

Double free

Aliasing + Mutation



Limitations of Other Languages

Java, Python, Ruby, C#, Scala, Go…

- Restrict direct access to memory.
- Run-time management of memory via periodic garbage collection.
- No explicit malloc and free, no memory corruption issues.
- But,
  - Overhead of tracking object references.
  - Program behavior unpredictable due to GC (bad for real-time systems).
  - Limited concurrency (global interpreter lock typical).
  - Larger code size.
  - VM must often be included.
  - Needs more memory and CPU power (i.e. not bare-metal).

Requirements for Safe System Programs

Must be fast and have minimal runtime overhead.

Should support direct memory access, but be memory safe.

Rust Language

Rust is a system programming language barely on hardware.

- No runtime requirement (runs fast).
- Control over memory allocation/destruction.
- Guarantees memory safety.

Developed to address severe memory leakage and corruption bugs in Firefox.

- First stable release in 5/2015.

➔ Rust Programming Language
➔ The Rust Book
➔ The Cargo Book

Rust Overview

Performance, as with C:
- Rust compilation to object code for bare-metal performance.

But, supports memory safety.
- Programs dereference only previously allocated pointers that have not been freed.
- Out-of-bound array accesses not allowed.

With low overhead.
- Compiler checks to make sure rules for memory safety are followed.
- Zero-cost abstraction in managing memory (i.e. no garbage collection).

Via:
- Advanced type system.
- Ownership, borrowing, and lifetime concepts to prevent memory corruption issues.

But at a cost:
- Cognitive cost to programmers who must think more about rules for using memory and references as they program.

Group Projects

Paper Presentations

< Introduction />