Lecture 14: Low-Level Programming

1. Introduction and Context

- Focus: Low-level software development, primarily in C and C++.
- These languages are considered "low-level" compared to Python, JavaScript, etc.
- Most modern software is not written in C/C++, but these languages remain crucial for performance-critical and system-level work.
- Tools for low-level development (debugging, performance, security) are the most mature due to decades of evolution.
- Many techniques discussed here are not yet available in higher-level languages, but may propagate over time.
- **Low-level languages** provide direct access to memory and hardware, with minimal abstraction. **High-level languages** abstract away hardware details for ease of use and safety.

Feature	Low-Level (C/C++)	High-Level (Python, JS)
Memory Management	Manual (malloc/free)	Automatic (GC, no pointers)
Hardware Access	Direct (pointers, I/O)	Indirect/abstracted
Performance	High, predictable	Lower, less predictable
Safety	Low (UB, buffer overflows)	High (bounds checks, types)
Use Cases	OS, drivers, embedded	Web, scripting, data science

- **Assembly** is even lower-level than C/C++, providing a 1:1 mapping to machine instructions. C/C++ is often called "portable assembly."

2. Attitude Toward Debugging

- **Avoid using a debugger as a first resort.**
  - Using a debugger often means you've already failed to prevent a bug.
  - Debugging is human-intensive and inefficient compared to preventive techniques.
- **Alternative:** Use compiler features and static/dynamic analysis to catch bugs early.
- This attitude is especially helpful at the low level, but is good practice at any level.

Approach	Description	Example Tool/Method
Preventive	Catch bugs before they happen	static_assert, -Wall
Reactive	Find/fix bugs after they occur	gdb, printf, valgrind

Example:

- *Static analysis*: Compiler warns about uninitialized variable (`-Wall`).
- *Debugger*: You run the program, it crashes, and you use gdb to find the bug.

3. Compilers as Debugging and Security Tools

3.1 GCC and Clang

- **GCC** and **Clang** are the two major free compilers for C/C++.
- They are largely compatible, but some options differ in spelling or behavior.
- Most techniques discussed apply to both, but examples use GCC syntax.

3.2 Security-Related Compiler Options

3.2.1 `_FORTIFY_SOURCE`

- **Usage:** `-D_FORTIFY_SOURCE=2 -O2`
- **Purpose:** Replaces standard functions (e.g., `memcpy`) with safer, bounds-checked versions.
- **How it works:**
  - Compiler injects extra runtime checks for buffer overflows and similar bugs.
  - Example: `memcpy(dest, src, size)` will check that `dest` and `src` are valid and within bounds if possible.
- **Limitations:**
  - If the compiler cannot determine object sizes, it falls back to the standard (unchecked) version.
  - Not a guarantee of safety—just a partial defense.

Example:

```
char buf[8];
memcpy(buf, src, 16); // _FORTIFY_SOURCE detects overflow at runtime if possible
```

3.2.2 `-fstack-protector`

- **Purpose:** Defends against stack buffer overflows by inserting a "canary" value in the stack frame.
- **How it works:**
  - Canary is a random value placed before the return address.
  - On function return, the canary is checked. If it has changed, the program aborts.
  - Analogy: Like a canary in a coal mine—detects danger early.

```
|----------------------|  <-- High Address
|   Local Variables    |
|----------------------|
|      Canary Value    |  <--- Inserted by compiler
|----------------------|
|   Return Address     |
|----------------------|  <-- Low Address
```

The canary sits between local variables and the return address. If a buffer overflow overwrites the canary, the program aborts.

Step-by-Step: Buffer Overflow Detection with Canary

1. Function is called; stack frame is set up with local variables, canary, and return address.
2. A buffer overflow (e.g., writing past the end of an array) overwrites the canary value.
3. Before returning, the function checks the canary.
4. If the canary is changed, the program aborts immediately, preventing exploitation.

Real-World Analogy:
Imagine a sealed envelope with a tamper-evident sticker (the canary). If someone tries to open the envelope (overflow the buffer), the sticker is broken. When you receive the envelope, you check the sticker; if it's broken, you know not to trust the contents.

3.2.3 Control Flow Enforcement Technology (CET) / Shadow Stack

- **Usage:** `-fcf-protection`
- **Purpose:** Hardware-enforced control flow integrity using a shadow stack.
- **How it works:**
  - The shadow stack stores only return addresses, is inaccessible to user code, and is protected by the OS.
  - On function call: return address is pushed to both the normal and shadow stack.
  - On return: both are popped and compared. If they differ, the program traps/crashes.

```
Normal Stack (accessible):
|----------------------|  <-- High Address
|   Local Variables    |
|----------------------|
|   Return Address     |  <--- Target for attacks
|----------------------|  <-- Low Address

Shadow Stack (protected):
|----------------------|  <-- High Address
|   Return Address     |  <--- Only return addresses, protected
|----------------------|  <-- Low Address
```

The shadow stack is protected and only manipulated by special instructions. On return, both stacks are checked for agreement.

Step-by-Step: Shadow Stack Operation

1. Function call: Return address is pushed to both stacks.
2. Function executes.
3. Function return: Return address is popped from both stacks and compared.
4. If mismatch, program aborts (attack detected).

Requirements:

- Supported on recent Intel/AMD CPUs (CET-capable).
- Requires OS and compiler support; not enabled by default everywhere.

Real-World Analogy:
Think of the shadow stack as a secure logbook kept in a locked safe. Every time you leave a room (function call), you write down your destination (return

address) in both your personal notebook (normal stack) and the logbook (shadow stack). When you return, both records must match. If someone tampers with your notebook but can't access the logbook, the mismatch is detected and an alarm sounds (program aborts).

Security Feature	Compiler Flag/Option	What It Protects Against
Buffer Overflow	`_FORTIFY_SOURCE`, `-fstack-protector`	Overwriting memory, stack attacks
Return Address Tampering	`-fcf-protection`	Control flow hijacking
Use of Dangerous APIs	Warnings, static analysis	Unsafe functions (gets, strcpy)

4. Performance Improvement via Compiler Features

4.1 Optimization Levels

Flag	Description	Speed	Size	Debuggability
-O0	No optimization (default)	Slow	Large	Best
-O1	Basic optimizations	Med	Med	Good
-O2	Moderate, safe optimizations (most common)	Fast	Med	OK
-O3	Aggressive optimizations; may be buggy	Fastest	Large	Poor
-Os	Optimize for size instead of speed	Fast	Small	OK
-Og	Optimize for debugging (intended, but often still confusing)	Med	Med	Good

- **Caveats:**
  - Higher levels (especially `-O3`, `-Ofast`) may introduce compiler bugs or make debugging harder.
  - Optimizations may reorder instructions, making debugging confusing.
  - `-Og` is intended for debuggability, but may still cause confusion.

Example of Optimization Bug:

```
volatile int x = 0;
if (x == 0) {
    // ...
}
// With -O3, compiler may optimize away code if it thinks x can't change.
```

4.2 Build Scripts and Makefiles

- Build systems (e.g., Makefiles) control how programs are compiled and linked.
- Makefiles specify dependencies and recipes for building each part of a program.
- C/C++ build systems are more mature and sophisticated than those for higher-level languages.

Simple Example Makefile:

```
all: main

main: main.o util.o
    gcc -o main main.o util.o

main.o: main.c util.h
    gcc -c main.c

util.o: util.c util.h
    gcc -c util.c

clean:
    rm -f *.o main
```

This Makefile builds an executable from two source files, tracking dependencies and providing a clean target.

Build Process with Makefile

```
[Source Files]
     |
     v
```

```
 [Makefile]
     |
     v
[Compiler (gcc/clang)]
     |
     v
[Object Files (.o)]
     |
     v
   [Linker]
     |
     v
[Executable]
```

The Makefile orchestrates the build process, specifying how to compile and link source files into an executable.

4.3 Link Time Optimization (LTO)

- **Usage:** `-flto`
- **How it works:**
  - Object files include extra information (intermediate representation of the source code).
  - At link time, the compiler can optimize across all modules, enabling inlining and other whole-program optimizations.
- **Advantages:**
  - Can produce smaller, faster code.
- **Disadvantages:**
  - Slower link times.
  - Higher risk of compiler bugs, especially for large programs.

Diagram: Normal vs. LTO Build

```
Normal Build:
[Source1.c] -> [Obj1.o] --\
[Source2.c] -> [Obj2.o] ---+--> [Linker] -> [Executable]

LTO Build:
[Source1.c] -> [Obj1.o (IR)] --\
[Source2.c] -> [Obj2.o (IR)] ---+--> [Linker (with IR)] -> [Optimized Executable]
```

4.4 Profile-Guided Optimization (PGO)

- **Usage:** `-fprofile-generate` and `-fprofile-use`
- **How it works:**
  1. Compile with `-fprofile-generate`.
  2. Run the program to collect profiling data.
  3. Recompile with `-fprofile-use` to optimize based on real usage.
- **Caveats:**
  - Test runs may not match production behavior, so optimizations may not always be ideal.

Example:

```
gcc -fprofile-generate -o myprog myprog.c
./myprog   # Run to collect data
gcc -fprofile-use -o myprog-opt myprog.c
```

4.5 Optimization Hints

4.5.1 `__builtin_unreachable()`

- **Purpose:** Tells the compiler that a code path cannot be reached.
- **Example:**

  ```
  if (a < 0) __builtin_unreachable();
  x = a / 1024; // Compiler can optimize to x = a >> 10;
  ```

- **Difference from `abort()`:**

    - `abort()` always crashes; `__builtin_unreachable()` allows the compiler to optimize more aggressively.
    - In debug mode, you can use `abort()`; in release mode, switch to `__builtin_unreachable()` for performance.
- **Caution:** Only use when you are certain the code is unreachable; otherwise, you risk undefined behavior.

4.5.2 Function Attributes: `cold` and `hot`

- **Usage:** `__attribute__((cold))` or `__attribute__((hot))`
- **Purpose:**
    - `cold`: Function is rarely called (e.g., error handling).
    - `hot`: Function is frequently called (e.g., main loop).
- **Effect:**
    - Compiler may place cold code away from hot code to improve cache usage.
    - Can improve performance by keeping hot code in the instruction cache.
- **Example:**

```
__attribute__((cold)) void error_log();
__attribute__((hot)) int main_loop();
```

4.5.3 Profiling and Coverage

- **Compiler can use profiling data to automatically determine hot/cold code.**
- **Caveat:** Profiling assumes test runs match production runs, which may not always be true.

5. Static and Dynamic Code Checking

5.1 Static Checking

- **Static checking:** Compiler checks code at compile time, before running the program.
- **Example:**

```
#include <assert.h>
static_assert(INT_MAX < UINT_MAX, "Int must be smaller than UInt");
```

- **Benefits:**
    - Catches bugs before the program runs.
    - No runtime overhead.
- **Limitations:**
    - Only works for properties that can be checked at compile time.

Analysis Type	When It Runs	Catches Bugs Before Run?	Overhead
Static	Compile time	Yes	None
Dynamic	Runtime	No (only if executed)	Yes

5.1.1 Compiler Warnings

- **Enable with:** `-Wall` (and other `-W` flags)
- **Common warnings:**

Flag	Description
`-Wall`	Enable commonly helpful warnings
`-Wparentheses`	Warn about ambiguous or confusing expressions
`-Waddress`	Warn about suspicious pointer comparisons
`-Wstrict-aliasing`	Warn about type-punning across incompatible pointers
`-Wmaybe-uninitialized`	Warn if a variable might be used without initialization
`-Wtype-limits`	Warn about always-true/false comparisons due to types

- **Examples:**
    - `if (a + b << c)` // May be ambiguous; parentheses may be needed.
    - `if (p == "abc")` // Comparing pointer to string literal's address.
    - `int v; if (n < 0) v = 3; if (n < -10) return v;` // v may be uninitialized.

5.1.2 Controversial Warnings

- Some warnings (e.g., `-Wstrict-aliasing`, `-Wtype-limits`) are controversial:
  - May produce false positives or be inappropriate for certain codebases (e.g., Linux kernel).
  - Developers may disable them for portability or practicality.

5.1.3 Interprocedural Analysis: `-fanalyzer`

- **Purpose:** Performs static analysis across function boundaries.
- **Pros:** Can find bugs that single-function analysis misses.
- **Cons:** Slow, may produce many false positives, not always practical for large codebases.

5.2 Function Attributes for Checking and Optimization

Attribute	Description	Example Usage
`noreturn`	Function does not return (e.g., `exit()`)	`void fatal() __attribute__((noreturn));`
`const`/`unsequenced`	No side-effects; return depends only on arguments	`int square(int) __attribute__((const));`
`pure`/`reproducible`	No side-effects; may read global state	`int get_val() __attribute__((pure));`
`hot`/`cold`	Indicate call frequency for optimization	`void err() __attribute__((cold));`

- **Benefits:**
  - Help compiler optimize and check code correctness.
  - Example: `noreturn` allows compiler to warn about unreachable code after `exit()`.

5.3 Dynamic Checking

- **Dynamic checking:** Compiler inserts runtime checks to catch errors as the program runs.
- **Trade-offs:**
  - Slows down execution.
  - Only catches bugs if the problematic code path is executed during testing.

Common Dynamic Tools:

Tool	What It Detects
AddressSanitizer	Buffer overflows, use-after-free
Valgrind	Memory leaks, invalid accesses
UndefinedBehaviorSanitizer	Undefined behavior

5.4 Safe Integer Arithmetic (C23)

- **Header:** `<stdckdint.h>`
- **Functions:** `ckd_add`, `ckd_sub`, etc.
- **Usage:**

```c
#include <stdckdint.h>
int r;
if (ckd_add(&r, a, b)) {
  // Overflow occurred
}
```

- **Purpose:** Reliably detect integer overflow, which is a common attack vector.
- **Available as of C23.**

Operation	Traditional C (Risk)	C23 Checked (Safe)
Addition	`int c = a + b;`	`ckd_add(&c, a, b);`
Subtraction	`int c = a - b;`	`ckd_sub(&c, a, b);`
Multiplication	`int c = a * b;`	`ckd_mul(&c, a, b);`

6. Summary Table: Compiler Tools for Reliability and Performance

Use Case	Recommended Tool/Option

Use Case	Recommended Tool/Option
Undefined Behavior	`-fsanitize`, `-Wall`, `static_assert`
Stack Protection	`-fstack-protector`, Shadow Stacks (CET)
Buffer Overflow	`_FORTIFY_SOURCE`, `-fsanitize=address`
Performance	`-O2`, `-O3`, PGO, `-flto`
Integer Overflow	`<stdckdint.h>`
Optimization hints	`__builtin_unreachable()`, `__attribute__((cold))`

7. Final Notes

- Low-level programming requires careful attention to security, performance, and correctness.
- Modern compilers provide a wealth of options and features to help developers write safer, faster, and more reliable code.
- Avoid relying solely on debuggers; leverage compiler checks, static/dynamic analysis, and modern language features.
- Stay up to date with new standards (e.g., C23) and hardware features (e.g., CET).

Summary Table: Key Compiler Security Features

Feature/Flag	Purpose	How It Protects
`_FORTIFY_SOURCE`	Bounds-checks standard functions	Detects buffer overflows at runtime
`-fstack-protector`	Inserts canary for stack protection	Detects stack buffer overflows
`-fcf-protection`	Enables shadow stack (CET)	Prevents return address tampering
`-flto`	Link Time Optimization	Enables cross-module optimization
`-fprofile-generate`/`-fprofile-use`	Profile-Guided Optimization	Optimizes based on real usage
`-Wall`	Enables common warnings	Catches likely bugs at compile time