Software Engineering

Lecture 10 – Error Handling & Debugging

© 2015-19 Dr. Florian Echtler
Bauhaus-Universität Weimar
<florian.echtler@uni-weimar.de>

Error handling with Exceptions

- Goal: better structure for error handling
- Examples in C(++), but applicable to most other languages
- Exceptions are *thrown* (error reporting) and *caught* (error handling)

Error handling in C

- somewhat messy, requires lots of clean-up code
- errors often indicated through return value
- NULL for pointers
- value < 0 for int, often stored in global variable `errno`

```
int process_files() {

    input1 = fopen(“file1”, “r”);
    if (input1 == NULL)
        return errno;

    input2 = fopen(“file2”, “r”);
    if (input2 == NULL) {
        fclose(input1);
        return errno;
    }

    input3 = fopen(“file3”, “r”);
    if (input3 == NULL) {
        fclose(input1);
        fclose(input2);
        return errno;
    }

    // process files
    [ … ]
    return 0;
}
```

Error handling in C (2)

- Alternative method using goto
- “Goto considered harmful” (E. Dijkstra, 1968)
- Code using goto generally harder to read/understand
- Possible exception as shown in example

```
int process_files() {

    input1 = fopen(“file1”, “r”);
    if (input1 == NULL)
        goto error_1;

    input2 = fopen(“file2”, “r”);
    if (input2 == NULL)
        goto error_2;

    input3 = fopen(“file3”, “r”);
    if (input3 == NULL)
        goto error_3;

    // process files
    [ … ]
    return 0;

error_3: fclose(input2);
error_2: fclose(input1);
error_1: return errno;
}
```

Error handling in C++

- Local objects destroyed at end of scope
- No need for explicit clean-up if properly implemented in destructor
- Difficult with non-local objects

```
int process_files() {

    MyFile input1(“file1”, “r”);
    if (!input1.valid())
        return -1;

    MyFile input2(“file2”, “r”);
    if (!input2.valid())
        return -1;

    MyFile input3(“file3”, “r”);
    If (!input3.valid())
        return -1;

    // process files
    [ … ]
    return 0;
}
```

Error handling in C++ (2)

- Solution: try/catch blocks for exceptions
- Classes from `std::` throw derivatives of `std::exception`
- `exception.what()` provides short text description of error

```
void process_files() {

  try {

    std::ifstream input1(“file1”);
    std::ifstream input2(“file2”);
    std::ifstream input3(“file3”);

    // process files
    [ … ]

  } catch (std::exception& e) {

    // display error message
    std::cout << e.what() << “\n”;

  }
}
```

Exceptions with `throw/try/catch`

- Multiple catch blocks for different types are possible
- Rule (C++): “Throw by value, catch by reference”
- Java: Throw anything derived from `java.lang.Throwable` (e.g. subclass of `RuntimeException`)

```
try {

    if (index < 0)
        throw std::runtime_error(
          “Index out of range” );

    if (index > 10000)
        throw -1;
}

// catch plain integer
catch (int e) {
    std::cout << “Error #“;
    std::cout << e << “\n”;
}

// catch base class
// of runtime_error
catch (std::exception& e) {
    std::cout << e.what() << “\n”;
}
```

Exception propagation

- Exceptions are propagated upwards until they are caught
- Use “catch-all” clauses for any type that does not have a specific handler
- Java catch-all:
  `catch (Exception ex)`

```
void throw_something() {
  // no try...catch here
  throw std::runtime_error();
}

void caller() {

  try {
    throw_something();
  }

  // exception caught here
  catch (std::exception& e) {
    std::cout << e.what() << “\n”;
  }

  // “catch-all” for any type
  catch (...) {
    std::cout << “default
      exception handler\n”;
  }
}
```

Debugging

- The “basic” approach: `printf()` & friends
  - Not very flexible, modifiable at compile-time only
  - Mental “reverse-engineering” of program state
  - Can slow program down (sometimes massively)
  - Advantages?
- The “smart” approach: debugger
  - Stop program at arbitrary point
  - Advance execution step-by-step
  - View arbitrary variables/objects

Debugger commands

Exact command names differ between debuggers (gdb, VS, XCode, …), functionality remains same

- Set Breakpoint

  → Execution stops at specified line or method
- Set Watchpoint

  → Execution stops when variable is changed

Debugger commands (2)

- Step → Execute a single line of code
  - Step Into: enter any called functions
  - Step Over: call functions as a whole and stop
- Inspect/Watch Variable
  → Value of variable is shown while stepping
- Run/Continue
  → Execute until breakpoint/watchpoint hit

Stack frames

- Stack: dedicated memory for each thread
  - keeps track of called functions and their local vars
  - grows *downward* as opposed to heap (malloc, new)
- Stack frame: local variables & return address of currently executed function
- Debugger can switch to earlier stack frames, i.e. inspect caller's context
- List of stack frames = *call stack*

Call stack

- Minimal example with up to 3 stack frames, state when execution paused in line 2

```
 1 int main() {
 2       printf("Hello!");
 3       func1();
 4      printf("Goodbye!");
 5 }

 6 void func1() {
 7       int i = 1;
 8       func2(i);
 9       i = 42;
10 }

11 void func2(int a) {
12       int b = a;
13 }
```

Start of stack:
Frame for main()

Call stack (2)

- Minimal example with up to 3 stack frames, state when execution paused in line 7

```
1 int main() {
2       printf(“Hello!”);
3       func1();
4      printf(“Goodbye!”);
5 }

6 void func1() {
7       int i = 1;
8       func2(i);
9       i = 42;
10 }

11 void func2(int a) {
12       int b = a;
13 }
```

```
Start of stack:
Frame for main()
Frame for func1()
    Return to main, line 4
    Storage for int i
```

Call stack (3)

- Minimal example with up to 3 stack frames, state when execution paused in line 12

```
 1 int main() {
 2       printf(“Hello!”);
 3       func1();
 4      printf(“Goodbye!”);
 5 }

 6 void func1() {
 7       int i = 1;
 8       func2(i);
 9       i = 42;
10 }

11 void func2(int a) {
12       int b = a;
13 }
```

```
Start of stack:
Frame for main()
Frame for func1()
    Return to main, line 4
    Storage for int i
Frame for func2()
    Return to func1, line 9
    Storage for int b
    Storage for int a
```

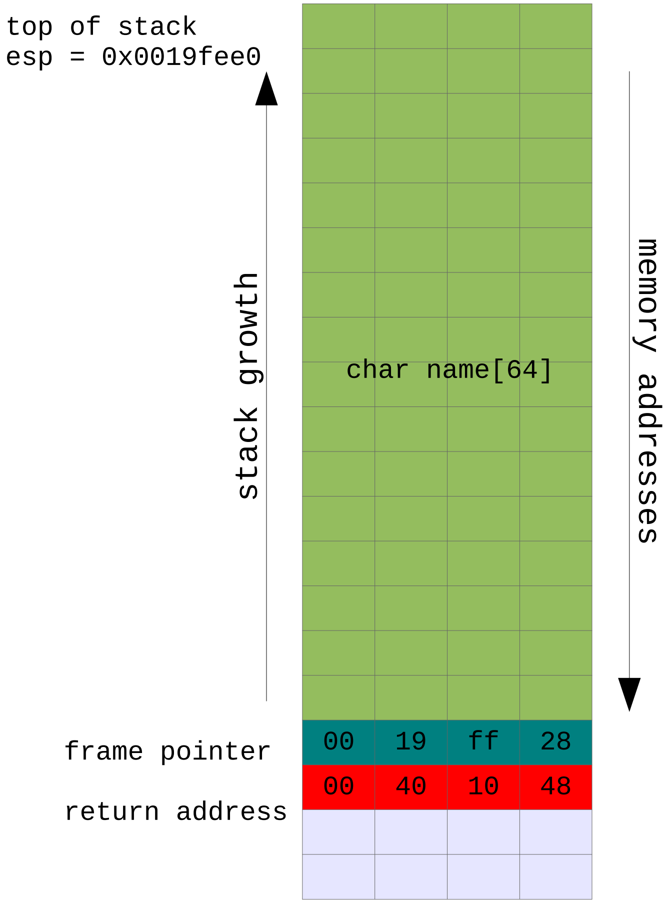
Debugging information

- Debugger does not know names of functions, variables etc.
  → can only show memory addrs
- “Human-readable” names must be compiled into executable, e.g. using `-g` flag (gcc/javac)
- Binary grows larger, sometimes slower
  → usually not enabled for release builds

Buffer Overflow (1)

Source (FU): http://arstechnica.com/security/...-the-buffer-overflow/

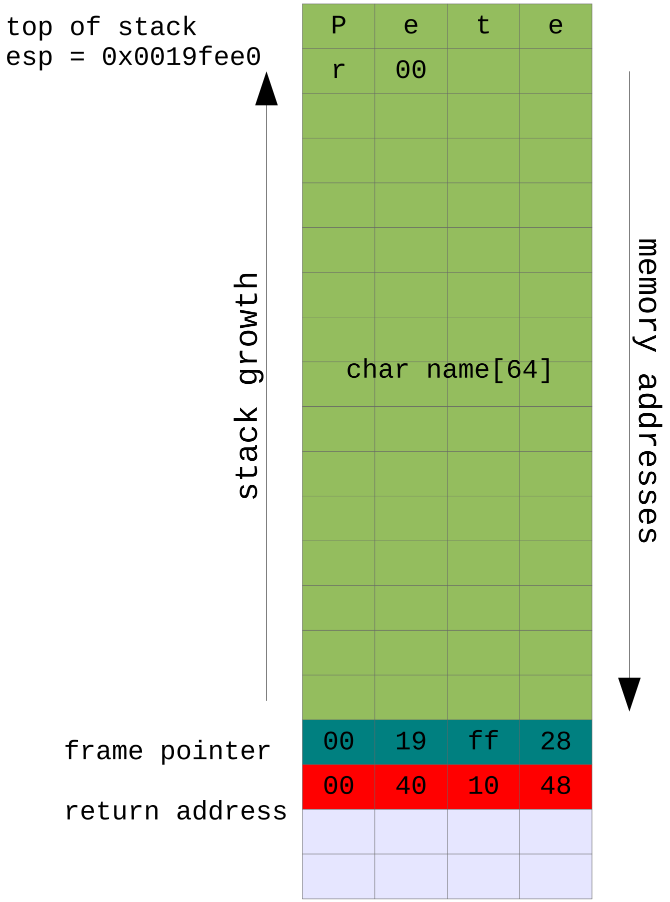
```
void get_name() {
    char name[64];
    printf(“Your name? ”);
    std::gets(name);
    printf(“Hi, %s.”,name);
}
```



Buffer Overflow (2)

Source (FU): http://arstechnica.com/security/...-the-buffer-overflow/

- `gets` = "Get String"
- Reads characters until newline
- Does *not* check any other conditions → problems?



Buffer Overflow (3)

Source (FU): http://arstechnica.com/security/...-the-buffer-overflow/

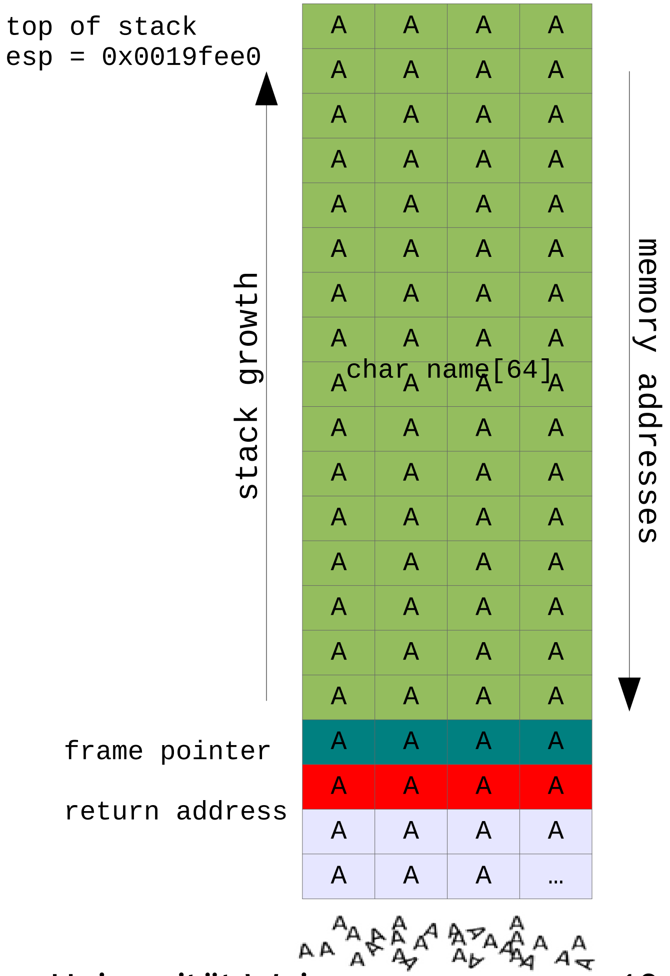
- *No size check!*

  → return address overwritten

  → program crashes

- Many similar, outdated C functions → listen to compiler warnings!

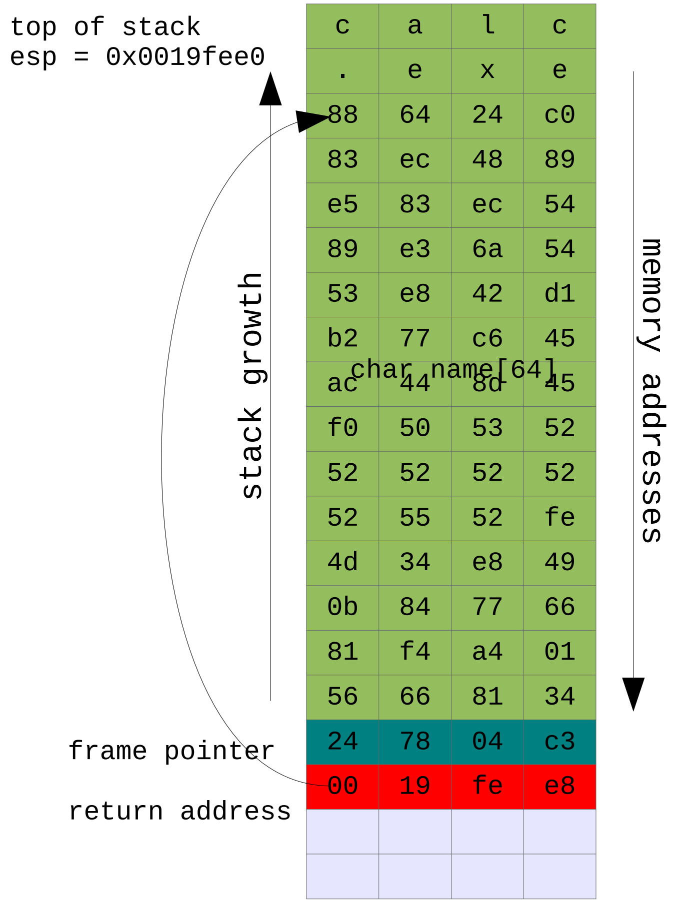
- Other problems?



Buffer Overflow (4)

Source (FU): http://arstechnica.com/security/...-the-buffer-overflow/

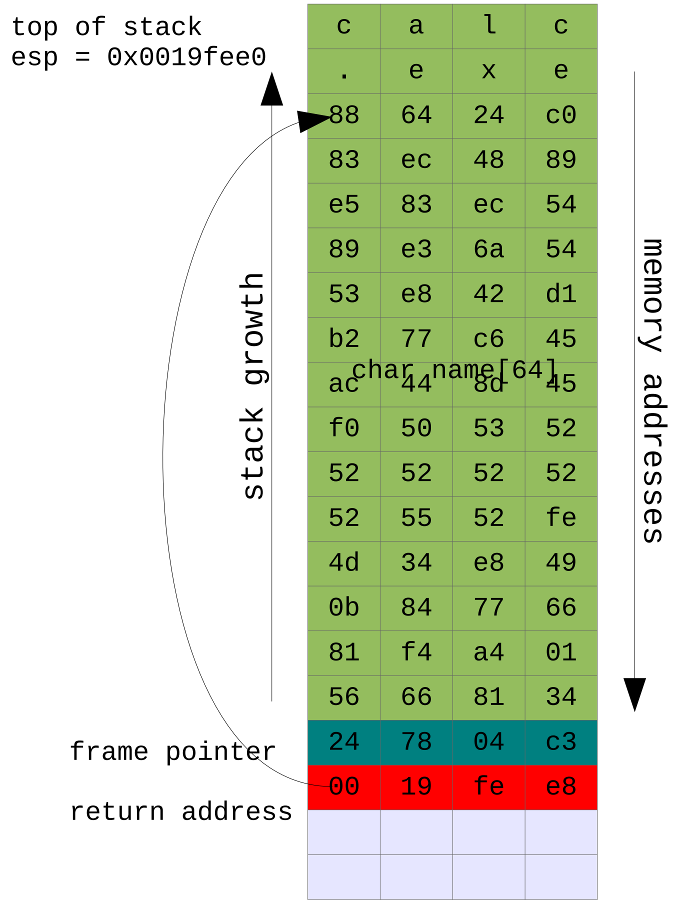
- Return address can be overwritten with pointer to buffer itself
- Possible to insert *and execute* arbitrary code!
- Root cause of many historic security issues (e.g. Morris worm)



Buffer Overflow (5)

Source (FU): http://arstechnica.com/security/...-the-buffer-overflow/

- Modern OS contain multiple countermeasures
- Possible mitigation strategies?
  - Address layout randomization
  - Stack “canary” values
  - No-Execute-Bit (NX)



Questions/Comments?