Secure Software Development – SSD

Assignment Defensive Programming

06.12.2023

Winter 2023/24, www.iaik.tugraz.at/ssd

Defensive Programming

Since you’re now an expert in fixing and exploiting bugs,
it is important to know how to avoid them.

Mistakes can happen everywhere

- Mistakes happen everywhere
- Especially in low-level C code
  - Look at the defenselets
- It is up to you to write better, safer code

Write Readable Code

- What does the following code do?
  `!ErrorHasOccured() ??!??! HandleError();`
- Error handling, but what is the `??!??!` operator?

```
#define MAGIC(e) (sizeof(struct { int:-!!(e); }))
```

- It is magic of course! What is `:-!!` though?
- Such code is unreadable and easily causes bugs

https://stackoverflow.com/questions/7825055/what-does-the-operator-do-in-c

https://stackoverflow.com/questions/9229601/what-is-in-c-code

https://stackoverflow.com/questions/652788/what-is-the-worst-real-world-macros

Goal

- Implement software in a secure manner
  - Use good coding style
  - Use defensive programming principles
  - Do proper error handling
  - Write your own tests
- Become a better software-engineer

Task: Defensive Programming

Defensive-Programming Part 1:

Deadline: 14th of January 23:59 (14.01.2024)

Tag: `defensive`

Next KU dates (possible to ask assignment questions):

13.12.2023 / 20.12.2023 / 10.01.2024

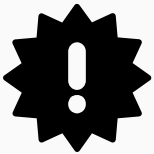
Defensive-Programming Part 1:

Deadline: 14th of January 23:59 (14.01.2024)

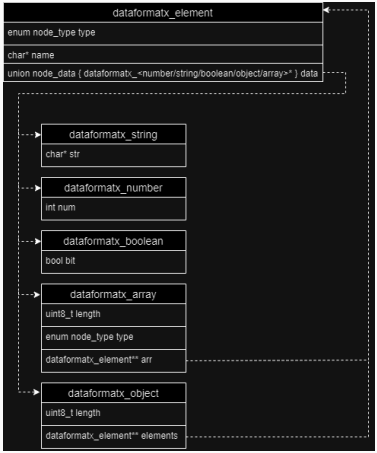
Tag: `defensive`

Next KU dates (possible to ask assignment questions):

13.12.2023 / 20.12.2023 / 10.01.2024

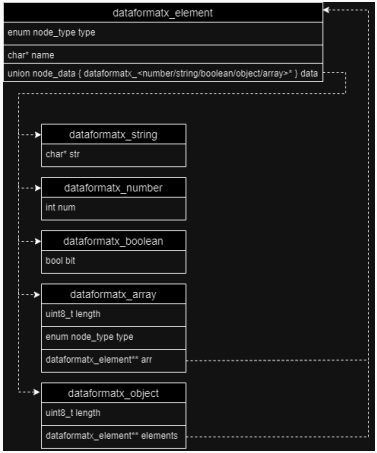


- Upstream: `https://extgit.iaik.tugraz.at/sase/practicals/2023/exercise2023-upstream.git`
  - `defensive/docker.sh`



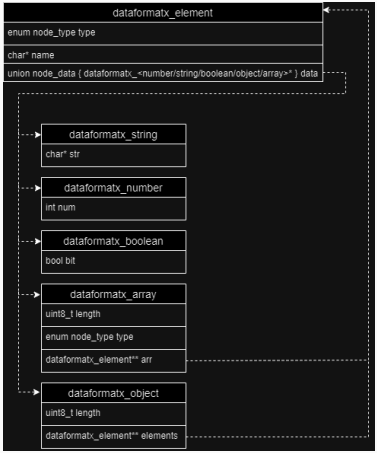
- Library to handle our own dataformat
  - Load/Write to/from files (already implemented)
  - Working with structure (add, remove, create, free nodes)
  - Handle values of nodes (set, get, merge, split, replace)
- Assignment: Implement remaining functionality

libdataformatx



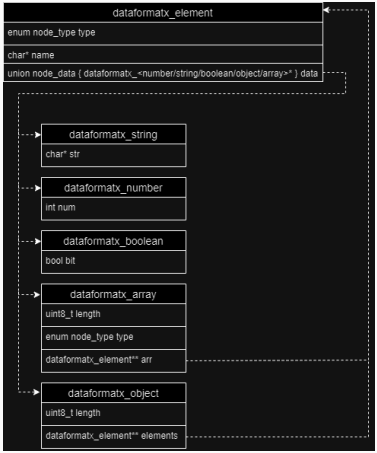
- Library to handle our own dataformat
  - Load/Write to/from files (already implemented)
  - Working with structure (add, remove, create, free nodes)
  - Handle values of nodes (set, get, merge, split, replace)
- Assignment: Implement remaining functionality

libdataformatx



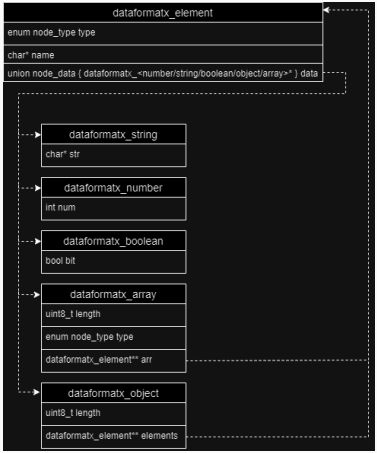
- Library to handle our own dataformat
  - Load/Write to/from files (already implemented)
  - Working with structure (add, remove, create, free nodes)
  - Handle values of nodes (set, get, merge, split, replace)
- Assignment: Implement remaining functionality

libdataformatx



- Library to handle our own dataformat
  - Load/Write to/from files (already implemented)
  - Working with structure (add, remove, create, free nodes)
  - Handle values of nodes (set, get, merge, split, replace)
- Assignment: Implement remaining functionality

libdataformatx



- Library to handle our own dataformat
  - Load/Write to/from files (already implemented)
  - Working with structure (add, remove, create, free nodes)
  - Handle values of nodes (set, get, merge, split, replace)
- Assignment: Implement remaining functionality

- We provide you with some basic implementation!
- Implement the remaining functions in a secure manner!
- Make sure the functions adhere to the documentation
- Use tools to find and fix implementation flaws!

- We provide you with some basic implementation!
- Implement the remaining functions in a secure manner!
- Make sure the functions adhere to the documentation
- Use tools to find and fix implementation flaws!

- We provide you with some basic implementation!
- Implement the remaining functions in a secure manner!
- Make sure the functions adhere to the documentation
- Use tools to find and fix implementation flaws!

- We provide you with some basic implementation!
- Implement the remaining functions in a secure manner!
- Make sure the functions adhere to the documentation
- Use tools to find and fix implementation flaws!

- How to find bugs:
  - Compiler warnings
  - Static code analysis (`cppcheck`/`scan-build`)
  - Valgrind, address-sanitizer, etc.
  - Look for inconsistencies with documentation (header files)
  - Test edge cases (integer overflows, out of memory, ...)
  - Fuzzing (e.g. AFL)

- How to find bugs:
  - Compiler warnings
  - Static code analysis (`cppcheck`/`scan-build`)
  - Valgrind, address-sanitizer, etc.
  - Look for inconsistencies with documentation (header files)
  - Test edge cases (integer overflows, out of memory, ...)
  - Fuzzing (e.g. AFL)

- How to find bugs:
  - Compiler warnings
  - Static code analysis (`cppcheck`/`scan-build`)
  - Valgrind, address-sanitizer, etc.
  - Look for inconsistencies with documentation (header files)
  - Test edge cases (integer overflows, out of memory, ...)
  - Fuzzing (e.g. AFL)

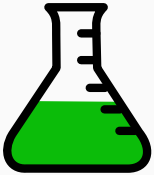
- How to find bugs:
  - Compiler warnings
  - Static code analysis (`cppcheck`/`scan-build`)
  - Valgrind, address-sanitizer, etc.
  - Look for inconsistencies with documentation (header files)
  - Test edge cases (integer overflows, out of memory, ...)
  - Fuzzing (e.g. AFL)

- How to find bugs:
  - Compiler warnings
  - Static code analysis (`cppcheck`/`scan-build`)
  - Valgrind, address-sanitizer, etc.
  - Look for inconsistencies with documentation (header files)
  - Test edge cases (integer overflows, out of memory, ...)
  - Fuzzing (e.g. AFL)

- How to find bugs:
  - Compiler warnings
  - Static code analysis (`cppcheck`/`scan-build`)
  - Valgrind, address-sanitizer, etc.
  - Look for inconsistencies with documentation (header files)
  - Test edge cases (integer overflows, out of memory, ...)
  - Fuzzing (e.g. AFL)

- How to find bugs:
  - Compiler warnings
  - Static code analysis (`cppcheck`/`scan-build`)
  - Valgrind, address-sanitizer, etc.
  - Look for inconsistencies with documentation (header files)
  - Test edge cases (integer overflows, out of memory, ...)
  - Fuzzing (e.g. AFL)

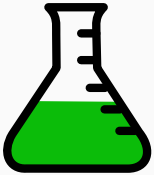
Test cases



- We provide some basic test cases
- They are correct but do not cover all edge cases!
- Implement your own exhaustive test cases
- Think of corner cases
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

Overall branch coverage	Bonus points
65% <= *cov* < 70%	1
70% <= *cov* < 80%	2
80% <= *cov* < 90%	3
90% <= *cov* < 95%	4
95% <= *cov*	5

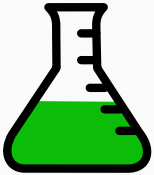
Test cases



- We provide some basic test cases
- They are correct but do not cover all edge cases!
- Implement your own exhaustive test cases
- Think of corner cases
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

Overall branch coverage	Bonus points
65% <= *cov* < 70%	1
70% <= *cov* < 80%	2
80% <= *cov* < 90%	3
90% <= *cov* < 95%	4
95% <= *cov*	5

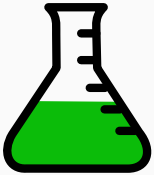
Test cases



- We provide some basic test cases
- They are correct but do not cover all edge cases!
- Implement your own exhaustive test cases
- Think of corner cases
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

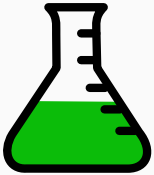
Overall branch coverage	Bonus points
65% <= *cov* < 70%	1
70% <= *cov* < 80%	2
80% <= *cov* < 90%	3
90% <= *cov* < 95%	4
95% <= *cov*	5

Test cases



- We provide some basic test cases
- They are correct but do not cover all edge cases!
- Implement your own exhaustive test cases
- Think of corner cases
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

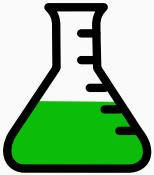
Overall branch coverage	Bonus points
65% <= *cov* < 70%	1
70% <= *cov* < 80%	2
80% <= *cov* < 90%	3
90% <= *cov* < 95%	4
95% <= *cov*	5



- We provide some basic test cases
- They are correct but do not cover all edge cases!
- Implement your own exhaustive test cases
- Think of corner cases
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

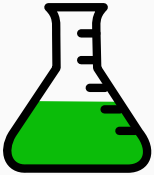
Overall branch coverage	Bonus points
65% <= *cov* < 70%	1
70% <= *cov* < 80%	2
80% <= *cov* < 90%	3
90% <= *cov* < 95%	4
95% <= *cov*	5

Test cases



- We provide some basic test cases
- They are correct but do not cover all edge cases!
- Implement your own exhaustive test cases
- Think of corner cases
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

Overall branch coverage	Bonus points
65% <= *cov* < 70%	1
70% <= *cov* < 80%	2
80% <= *cov* < 90%	3
90% <= *cov* < 95%	4
95% <= *cov*	5



- We provide some basic test cases
- They are correct but do not cover all edge cases!
- Implement your own exhaustive test cases
- Think of corner cases
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

Overall branch coverage	Bonus points
$65\% <= cov < 70\%$	1
$70\% <= cov < 80\%$	2
$80\% <= cov < 90\%$	3
$90\% <= cov < 95\%$	4
$95\% <= cov$	5

Earning Points

- 35 points for Defensive Part
  - Correctly implement all functions
- 5 bonus points
  - Bonus points are given for good code coverage.
  - Check using: make test && make gcov

Earning Points

- 35 points for Defensive Part
  - Correctly implement all functions
- 5 bonus points
  - Bonus points are given for good code coverage.
  - Check using: make test && make gcov

- 35 points for Defensive Part
  - Correctly implement all functions
- 5 bonus points
  - Bonus points are given for good code coverage.
  - Check using: make test && make gcov

We test your submission against our own test suite.

Implementation flaws or issues will (in addition to failed testcases) globally reduce points, regardless of whether exploitable or not!

- -5 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions/leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, e.g. `(void*)x + 1`
  - Non-portable, hidden assumptions, e.g. `sizeof(int) == 4`
  - Hard-to-read or dangerous code, e.g. `#define F(x) x = x*x`
  - Use of global variables
  - Compiler warnings with `-Wall`

We test your submission against our own test suite.

Implementation flaws or issues will (in addition to failed testcases) globally reduce points, regardless of whether exploitable or not!

- -5 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions/leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, e.g. `(void*)x + 1`
  - Non-portable, hidden assumptions, e.g. `sizeof(int) == 4`
  - Hard-to-read or dangerous code, e.g. `#define F(x) x = x*x`
  - Use of global variables
  - Compiler warnings with `-Wall`

We test your submission against our own test suite.

Implementation flaws or issues will (in addition to failed testcases) globally reduce points, regardless of whether exploitable or not!

- -5 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions/leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, e.g. `(void*)x + 1`
  - Non-portable, hidden assumptions, e.g. `sizeof(int) == 4`
  - Hard-to-read or dangerous code, e.g. `#define F(x) x = x*x`
  - Use of global variables
  - Compiler warnings with `-Wall`

We test your submission against our own test suite.

Implementation flaws or issues will (in addition to failed testcases) globally reduce points, regardless of whether exploitable or not!

- -5 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions/leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, e.g. `(void*)x + 1`
  - Non-portable, hidden assumptions, e.g. `sizeof(int) == 4`
  - Hard-to-read or dangerous code, e.g. `#define F(x) x = x*x`
  - Use of global variables
  - Compiler warnings with `-Wall`

We test your submission against our own test suite.

Implementation flaws or issues will (in addition to failed testcases) globally reduce points, regardless of whether exploitable or not!

- -5 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions/leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, e.g. `(void*)x + 1`
  - Non-portable, hidden assumptions, e.g. `sizeof(int) == 4`
  - Hard-to-read or dangerous code, e.g. `#define F(x) x = x*x`
  - Use of global variables
  - Compiler warnings with `-Wall`

We test your submission against our own test suite.

Implementation flaws or issues will (in addition to failed testcases) globally reduce points, regardless of whether exploitable or not!

- -5 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions/leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, e.g. `(void*)x + 1`
  - Non-portable, hidden assumptions, e.g. `sizeof(int) == 4`
  - Hard-to-read or dangerous code, e.g. `#define F(x) x = x*x`
  - Use of global variables
  - Compiler warnings with `-Wall`

We test your submission against our own test suite.

Implementation flaws or issues will (in addition to failed testcases) globally reduce points, regardless of whether exploitable or not!

- -5 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions/leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, e.g. `(void*)x + 1`
  - Non-portable, hidden assumptions, e.g. `sizeof(int) == 4`
  - Hard-to-read or dangerous code, e.g. `#define F(x) x = x*x`
  - Use of global variables
  - Compiler warnings with `-Wall`

We test your submission against our own test suite.

Implementation flaws or issues will (in addition to failed testcases) globally reduce points, regardless of whether exploitable or not!

- -5 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions/leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, e.g. `(void*)x + 1`
  - Non-portable, hidden assumptions, e.g. `sizeof(int) == 4`
  - Hard-to-read or dangerous code, e.g. `#define F(x) x = x*x`
  - Use of global variables
  - Compiler warnings with `-Wall`

We test your submission against our own test suite.

Implementation flaws or issues will (in addition to failed testcases) globally reduce points, regardless of whether exploitable or not!

- -5 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions/leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, e.g. `(void*)x + 1`
  - Non-portable, hidden assumptions, e.g. `sizeof(int) == 4`
  - Hard-to-read or dangerous code, e.g. `#define F(x) x = x*x`
  - Use of global variables
  - Compiler warnings with `-Wall`

We test your submission against our own test suite.

Implementation flaws or issues will (in addition to failed testcases) globally reduce points, regardless of whether exploitable or not!

- -5 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions/leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, e.g. `(void*)x + 1`
  - Non-portable, hidden assumptions, e.g. `sizeof(int) == 4`
  - Hard-to-read or dangerous code, e.g. `#define F(x) x = x*x`
  - Use of global variables
  - Compiler warnings with `-Wall`

HELP! How should I start?

- Pull from upstream
- Read the provided `README.md`, `Assignment.md`
- Read the provided header files
- Check out already implemented functions
- Ask on our Discord channel!
- Come by during question hours!

HELP! How should I start?

- Pull from upstream
- Read the provided `README.md`, `Assignment.md`
- Read the provided header files
- Check out already implemented functions
- Ask on our Discord channel!
- Come by during question hours!

HELP! How should I start?

- Pull from upstream
- Read the provided `README.md`, `Assignment.md`
- Read the provided header files
- Check out already implemented functions
- Ask on our Discord channel!
- Come by during question hours!

HELP! How should I start?

- Pull from upstream
- Read the provided `README.md`, `Assignment.md`
- Read the provided header files
- Check out already implemented functions
- Ask on our Discord channel!
- Come by during question hours!

- Pull from upstream
- Read the provided `README.md`, `Assignment.md`
- Read the provided header files
- Check out already implemented functions
- Ask on our Discord channel!
- Come by during question hours!

HELP! How should I start?

- Pull from upstream
- Read the provided `README.md`, `Assignment.md`
- Read the provided header files
- Check out already implemented functions
- Ask on our Discord channel!
- Come by during question hours!