Secure Software Development

Assignment D1+D2: Defensive Programming

Ehrenreich, Hadzic, Lamster, Nageler, Schwarzl, Weiser

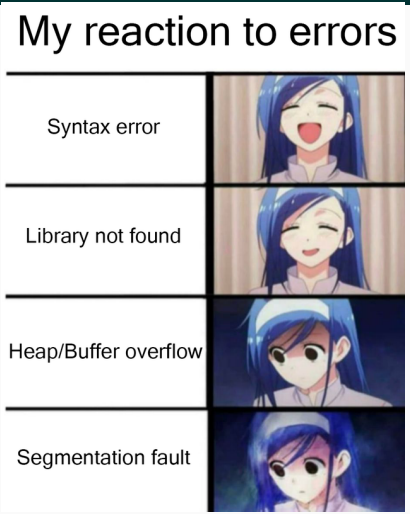
11.11.2020

Winter 2020/21, www.iaik.tugraz.at/ssd

Defensive Programming

- Deadline D1: **11.12.2020 - 23:59:59** tag: **d1**
- Deadline D2: **18.12.2020 - 23:59:59** tag: **d2**

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



- Mistakes happen everywhere (especially in C)
  - Look at the hacklets from assignments H1+H2
- It is up to you to write better, safer code

Write Readable Code

- What does the following code do?

```
!ErrorHasOccured() ??!??! HandleError();
```

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

Write Readable Code

- What does the following code do?

```
!ErrorHasOccured() ??!??! HandleError();
```

- Error handling, but what is the ??!??! operator?
- What about this beautiful #define?

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

- What does the following code do?
  ```
  !ErrorHasOccured() ??!??! HandleError();
  ```
- Error handling, but what is the `??!??!` operator?
- What about this beautiful `#define`?
  ```
  #define MAGIC(e) (sizeof(struct { int:-!!(e); }))
  ```
- It is magic of course! What is `:-!!` though?

- What does the following code do?
  ```
  !ErrorHasOccured() ??!??! HandleError();
  ```
- Error handling, but what is the `??!??!` operator?
- What about this beautiful `#define`?
  ```
  #define MAGIC(e) (sizeof(struct { int:-!!(e); }))
  ```
- It is magic of course! What is `:-!!` though?
- Such code is unreadable and easily causes bugs

Write Readable Code

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

General goal

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

General 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: S5

- Implement a *delicious* library called libs5

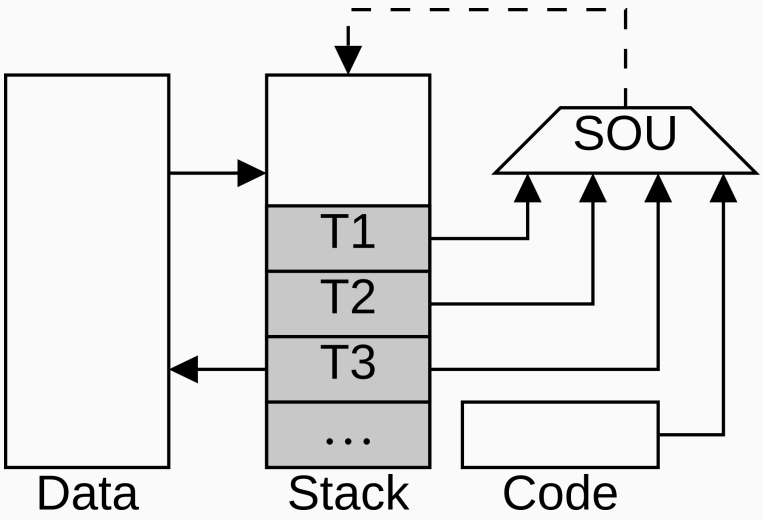
- Implement a *delicious* library called libs5
- S5 is a stack machine for string operations

- Implement a *delicious* library called libs5
- S5 is a stack machine for string operations
  - Data memory and stack memory

- Implement a *delicious* library called libs5
- S5 is a stack machine for string operations
  - Data memory and stack memory
  - String operation unit (SOU)

- Implement a *delicious* library called libs5
- S5 is a stack machine for string operations
  - Data memory and stack memory
  - String operation unit (SOU)
  - String operation commands

- Implement a *delicious* library called libs5
- S5 is a stack machine for string operations
  - Data memory and stack memory
  - String operation unit (SOU)
  - String operation commands
- Parse and execute S5 programs using libs5


SOU
T1
T2
T3
...
Data
Stack
Code

- S5 must support basic internal memory handling:

Operations with S5

- S5 must support basic internal memory handling:
  - Init, Delete, Pop, Push, Store, Load, Getters, Setters

- S5 must support basic internal memory handling:
  - Init, Delete, Pop, Push, Store, Load, Getters, Setters
  - These are used to implement everything else!

- S5 must support basic internal memory handling:
  - Init, Delete, Pop, Push, Store, Load, Getters, Setters
  - These are used to implement everything else!
- S5 should support the following *stack* operations:

- S5 must support basic internal memory handling:
  - Init, Delete, Pop, Push, Store, Load, Getters, Setters
  - These are used to implement everything else!
- S5 should support the following *stack* operations:
  - Store (`!`), Load (`@`), Drop (`0`), Dup (`2`), Over (`/`)

- S5 must support basic internal memory handling:
  - Init, Delete, Pop, Push, Store, Load, Getters, Setters
  - These are used to implement everything else!
- S5 should support the following *stack* operations:
  - Store (`!`), Load (`@`), Drop (`0`), Dup (`2`), Over (`/`)
- S5 should support the following *string* operations:

- S5 must support basic internal memory handling:
  - Init, Delete, Pop, Push, Store, Load, Getters, Setters
  - These are used to implement everything else!
- S5 should support the following *stack* operations:
  - Store (`!`), Load (`@`), Drop (`0`), Dup (`2`), Over (`/`)
- S5 should support the following *string* operations:
  - Reverse (`~`), Insert (`^`), Slice (`:`), Split (`*`), Replace (`%`)

Example Program


```
3                ; three memory locations
 hello           ; mem[0]
world            ; mem[1]
                 ; mem[2]
7                ; eight instructions
@ 0              ; T1 = mem[0]
~                ; reverse T1
! 0              ; mem[0] = T1, T1 removed
@ 1              ; T1 = mem[1]
@ 0              ; T2 = T1, T1 = mem[0]
^ 0              ; T1 = insert T1 into T2 at 0
! 2              ; mem[2] = T1, T1 removed
```

Defensive 1:

Defensive 1:

- 100 regular points (25%)
  - 0 points per internal function
  - 4 points per stack command
  - 12 points per string command
  - 20 points for file parsing

Defensive 1:

- 100 regular points (25%)
  - 0 points per internal function
  - 4 points per stack command
  - 12 points per string command
  - 20 points for file parsing
- 20 bonus points (5%) for code coverage

Defensive 1:

- 100 regular points (25%)
  - 0 points per internal function
  - 4 points per stack command
  - 12 points per string command
  - 20 points for file parsing
- 20 bonus points (5%) for code coverage
- See the API description in `s5.h`

Defensive 1:

- 100 regular points (25%)
  - 0 points per internal function
  - 4 points per stack command
  - 12 points per string command
  - 20 points for file parsing
- 20 bonus points (5%) for code coverage
- See the API description in `s5.h`

Defensive 2:

Defensive 1:

- 100 regular points (25%)
  - 0 points per internal function
  - 4 points per stack command
  - 12 points per string command
  - 20 points for file parsing
- 20 bonus points (5%) for code coverage
- See the API description in `s5.h`

Defensive 2:

- 84 regular points (21%)
  - 0 points per internal function
  - 4 points per stack command
  - 6 points per string command
  - 34 points for file parsing

Defensive 1:

- 100 regular points (25%)
  - 0 points per internal function
  - 4 points per stack command
  - 12 points per string command
  - 20 points for file parsing
- 20 bonus points (5%) for code coverage
- See the API description in `s5.h`

Defensive 2:

- 84 regular points (21%)
  - 0 points per internal function
  - 4 points per stack command
  - 6 points per string command
  - 34 points for file parsing
- API is pretty much the same

In our testing framework, some functions must work so that others can be tested. Here is a full overview of how these dependencies look:

ID	Function Name	Dependencies
A	Init	
B	Delete	A
C	Pop / Push / Store / Load	A B
E	Set Memory	A B
F	Get Memory / Size	A B C E
G	Get Stack / Pos	A B C E
H	Cmd Store / Cmd Load	A B E
I	Cmd Drop / Cmd Dup	A B C G
K	Cmd Over / Cmd Reverse	A B C
M	Cmd Insert / Cmd Slice	A B C
O	Cmd Split / Cmd Replace	A B C
Q	File Parsing	All

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

- -3 points per issue

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

- -3 points per issue
  - Hard program crash, segfault and similar

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

- -3 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions and leaks, use after free, use of uninitialized memory

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

- -3 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions and leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co

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

- -3 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions and leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...

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

- -3 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions and leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, non-portable code

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

- -3 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions and leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, non-portable code
  - Hard-to-read and dangerous code, i.e. `#define`

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

- -3 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions and leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, non-portable code
  - Hard-to-read and dangerous code, i.e. `#define`
  - Use of global variables

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

- -3 points per issue
  - Hard program crash, segfault and similar
  - Memory corruptions and leaks, use after free, use of uninitialized memory
  - other stuff reported by valgrind, address sanitizer & co
  - Format string vulnerability, integer overflow, ...
  - Undefined behavior, non-portable code
  - Hard-to-read and dangerous code, i.e. `#define`
  - Use of global variables
  - Compiler warnings with `-Wall`

- We test your submission against our own test suite

- We test your submission against our own test suite
- Here is how you can avoid bugs:

Implementation Flaws II

- We test your submission against our own test suite
- Here is how you can avoid bugs:
  - Listen to your compiler and eliminate warnings

- We test your submission against our own test suite
- Here is how you can avoid bugs:
  - Listen to your compiler and eliminate warnings
  - Use static code analysis like scan-build

- We test your submission against our own test suite
- Here is how you can avoid bugs:
  - Listen to your compiler and eliminate warnings
  - Use static code analysis like scan-build
  - Use a fuzzing framework like AFL

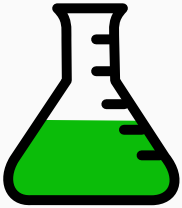
- We test your submission against our own test suite
- Here is how you can avoid bugs:
  - Listen to your compiler and eliminate warnings
  - Use static code analysis like scan-build
  - Use a fuzzing framework like AFL
  - Write your own test cases

- We test your submission against our own test suite
- Here is how you can avoid bugs:
  - Listen to your compiler and eliminate warnings
  - Use static code analysis like scan-build
  - Use a fuzzing framework like AFL
  - Write your own test cases
  - Use valgrind, address-sanitizer, etc.

- We test your submission against our own test suite
- Here is how you can avoid bugs:
  - Listen to your compiler and eliminate warnings
  - Use static code analysis like scan-build
  - Use a fuzzing framework like AFL
  - Write your own test cases
  - Use valgrind, address-sanitizer, etc.
  - Let your experienced colleagues check your code ☺

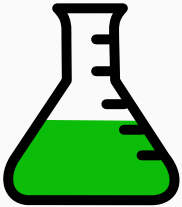
- We test your submission against our own test suite
- Here is how you can avoid bugs:
  - Listen to your compiler and eliminate warnings
  - Use static code analysis like scan-build
  - Use a fuzzing framework like AFL
  - Write your own test cases
  - Use valgrind, address-sanitizer, etc.
  - Let your experienced colleagues check your code ☺
- Reuse code when possible and avoid duplication

- Implement your own exhaustive test cases



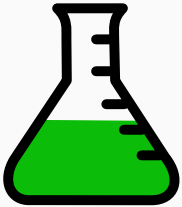
Test cases

- Implement your own exhaustive test cases
- Think of corner cases



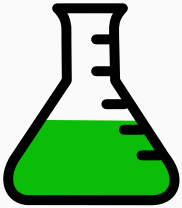
Test cases

- Implement your own exhaustive test cases
- Think of corner cases
  - Popping from an empty stack

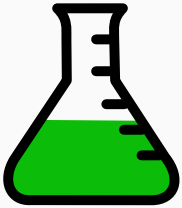


Test cases

- Implement your own exhaustive test cases
- Think of corner cases
  - Popping from an empty stack
  - Stack size exhausted

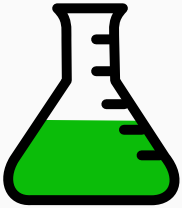


Test cases



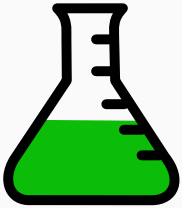
- Implement your own exhaustive test cases
- Think of corner cases
  - Popping from an empty stack
  - Stack size exhausted
  - String out-of-bounds

Test cases



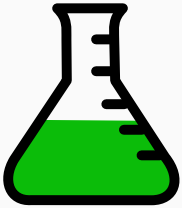
- Implement your own exhaustive test cases
- Think of corner cases
  - Popping from an empty stack
  - Stack size exhausted
  - String out-of-bounds
  - Unexpected formatting of input files

Test cases



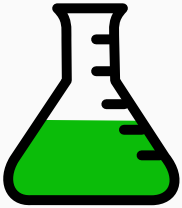
- Implement your own exhaustive test cases
- Think of corner cases
  - Popping from an empty stack
  - Stack size exhausted
  - String out-of-bounds
  - Unexpected formatting of input files
  - NULL pointers, integer overflows

Test cases

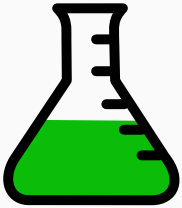


- Implement your own exhaustive test cases
- Think of corner cases
  - Popping from an empty stack
  - Stack size exhausted
  - String out-of-bounds
  - Unexpected formatting of input files
  - NULL pointers, integer overflows
  - Out of memory, file reading failed

Test cases



- Implement your own exhaustive test cases
- Think of corner cases
  - Popping from an empty stack
  - Stack size exhausted
  - String out-of-bounds
  - Unexpected formatting of input files
  - NULL pointers, integer overflows
  - Out of memory, file reading failed
- Good coverage yields bonus points (if above 50%)



- Implement your own exhaustive test cases
- Think of corner cases
  - Popping from an empty stack
  - Stack size exhausted
  - String out-of-bounds
  - Unexpected formatting of input files
  - NULL pointers, integer overflows
  - Out of memory, file reading failed
- Good coverage yields bonus points (if above 50%)

Overall branch coverage	Bonus points
$75\% <= cov < 80\%$	4 (1%)
$80\% <= cov < 85\%$	8 (2%)
$85\% <= cov < 90\%$	12 (3%)
$90\% <= cov < 95\%$	16 (4%)
$95\% <= cov$	20 (5%)

HELP! How should I start?

- Look through the provided `readme` and `s5.h`

HELP! How should I start?

- Look through the provided `readme` and `s5.h`
- Understand the API contracts we provided:

HELP! How should I start?

- Look through the provided `readme` and `s5.h`
- Understand the API contracts we provided:
  - Who owns shared data, who can modify it

HELP! How should I start?

- Look through the provided `readme` and `s5.h`
- Understand the API contracts we provided:
  - Who owns shared data, who can modify it
  - What preconditions are required

HELP! How should I start?

- Look through the provided `readme` and `s5.h`
- Understand the API contracts we provided:
  - Who owns shared data, who can modify it
  - What preconditions are required
- Each documentation string provides an example of what should happen when a given command is executed

HELP! How should I start?

- Look through the provided `readme` and `s5.h`
- Understand the API contracts we provided:
  - Who owns shared data, who can modify it
  - What preconditions are required
- Each documentation string provides an example of what should happen when a given command is executed
- Ask around on our Discord channel!

HELP! How should I start?

- Look through the provided `readme` and `s5.h`
- Understand the API contracts we provided:
  - Who owns shared data, who can modify it
  - What preconditions are required
- Each documentation string provides an example of what should happen when a given command is executed
- Ask around on our Discord channel!
- Come by during question hours!

Any Questions?