Secure Software Development – SSD

Assignment Defensive Programming

Schrammel, Koschatko

10.11.2021

Winter 2021/22, www.iaik.tugraz.at/ssd

Defensive Programming

Since you’re now an expert in 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

Timeline

Defensive-Programming:

Deadline: 7th of January 23:59 (07.01.2022)

Tag: `defensive`

Question Hour:

1st of December (01.12.2021)

Timeline

Defensive-Programming:

Deadline: 7th of January 23:59 (07.01.2022)

Tag: `defensive`

Question Hour:

1st of December (01.12.2021)

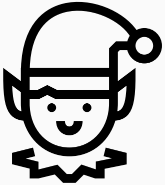
Links

- Test System: `https://sase.student.iaik.tugraz.at/`
- Upstream: `https://extgit.iaik.tugraz.at/sase/practicals/2021/exercise2021-upstream.git`
    - `defensive/docker.sh`

Links

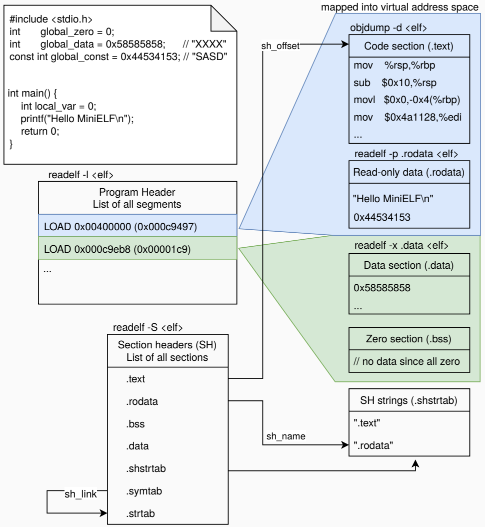
- Test System: `https://sase.student.iaik.tugraz.at/`
- Upstream: `https://extgit.iaik.tugraz.at/sase/practicals/2021/exercise2021-upstream.git`
  - `defensive/docker.sh`

MiniELF



- Implement your own library called libmelf
- $\mathrm{MiniELF} \subset \mathrm{ELF}$ (Executable and Linkable Format)
  - Parse existing ELF file
  - Access and modify it
  - Write new ELF file

MiniELF



```
#include <stdio.h>
int      global_zero = 0;
int      global_data = 0x58585858;     // "XXXX"
const int global_const = 0x44534153; // "SASD"


int main() {
    int local_var = 0;
    printf("Hello MiniELF\n");
    return 0;
}
```

```
mov    %rsp,%rbp
sub    $0x10,%rsp
movl   $0x0,-0x4(%rbp)
mov    $0x4a1128,%edi
...
```

MiniELF vs ELF

- You only need a subset of ELF
  - Static ELF binaries (executables and object files)
  - No overlapping sections/segments
  - Most important sections
    - `.text`, `.data`, `.rodata`, `.bss`, `.shstrtab`
- In particular, you do not need special treatment of:
  - Dynamic binaries, etc.: `.strtab`, `.symtab`, `.dyn*`, `.rela`, `.plt`, `.got`, `.jcr`, `.tdata`, `.tbss`, `.tcommon`, `.debug*`, `.note*`, `.gnu*`, `.comment`, ...

MiniELF vs ELF

- You only need a subset of ELF
  - Static ELF binaries (executables and object files)
  - No overlapping sections/segments
  - Most important sections
    - `.text`, `.data`, `.rodata`, `.bss`, `.shstrtab`
- In particular, you do not need special treatment of:
  - Dynamic binaries, etc.: `.strtab`, `.symtab`, `.dyn*`, `.rela`, `.plt`, `.got`, `.jcr`, `.tdata`, `.tbss`, `.tcommon`, `.debug*`, `.note*`, `.gnu*`, `.comment`, ...

MiniELF vs ELF

- You only need a subset of ELF
  - Static ELF binaries (executables and object files)
  - No overlapping sections/segments
  - Most important sections
    - `.text`, `.data`, `.rodata`, `.bss`, `.shstrtab`
- In particular, you do not need special treatment of:
  - Dynamic binaries, etc.: `.strtab`, `.symtab`, `.dyn*`, `.rela`, `.plt`, `.got`, `.jcr`, `.tdata`, `.tbss`, `.tcommon`, `.debug*`, `.note*`, `.gnu*`, `.comment`, ...

MiniELF vs ELF

- You only need a subset of ELF
  - Static ELF binaries (executables and object files)
  - No overlapping sections/segments
  - Most important sections
    - `.text`, `.data`, `.rodata`, `.bss`, `.shstrtab`
- In particular, you do not need special treatment of:
  - Dynamic binaries, etc.: `.strtab`, `.symtab`, `.dyn*`, `.rela`, `.plt`, `.got`, `.jcr`, `.tdata`, `.tbss`, `.tcommon`, `.debug*`, `.note*`, `.gnu*`, `.comment`, ...

MiniELF vs ELF

- You only need a subset of ELF
  - Static ELF binaries (executables and object files)
  - No overlapping sections/segments
  - Most important sections
    - `.text`, `.data`, `.rodata`, `.bss`, `.shstrtab`
- In particular, you do not need special treatment of:
  - Dynamic binaries, etc.: `.strtab`, `.symtab`, `.dyn*`, `.rela`, `.plt`, `.got`, `.jcr`, `.tdata`, `.tbss`, `.tcommon`, `.debug*`, `.note*`, `.gnu*`, `.comment`, ...

Earning Points

- 100 regular points
  - open ELF file
  - read sections + segments
  - modify sections + segments
  - write ELF file
- 20 bonus points
  - code coverage

Earning Points

- 100 regular points
  - open ELF file
  - read sections + segments
  - modify sections + segments
  - write ELF file
- 20 bonus points
  - code coverage

Functional Dependencies

In our testing framework, some functions must work so that others can be tested.

E.g.,

- Most API functions and test-cases require `libmelf_open`
- `libmelf_setSegmentData` may require `libmelf_getSegmentData`
- ...

Implementation Flaws I

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`

Implementation Flaws I

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`

Implementation Flaws I

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`

Implementation Flaws I

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`

Implementation Flaws I

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`

Implementation Flaws I

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`

Implementation Flaws I

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`

Implementation Flaws I

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`

Implementation Flaws I

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`

Implementation Flaws I

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`

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
  - Write your own test cases
  - Use static code analysis like *cppcheck* or *scan-build*
  - Use a fuzzing framework like AFL
  - Use valgrind, address-sanitizer, etc.
  - Let your experienced colleagues check your code ☺
- Reuse code when possible and avoid duplication

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
  - Write your own test cases
  - Use static code analysis like *cppcheck* or *scan-build*
  - Use a fuzzing framework like AFL
  - Use valgrind, address-sanitizer, etc.
  - Let your experienced colleagues check your code ☺
- Reuse code when possible and avoid duplication

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
  - Write your own test cases
  - Use static code analysis like *cppcheck* or *scan-build*
  - Use a fuzzing framework like AFL
  - Use valgrind, address-sanitizer, etc.
  - Let your experienced colleagues check your code ☺
- Reuse code when possible and avoid duplication

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
  - Write your own test cases
  - Use static code analysis like *cppcheck* or *scan-build*
  - Use a fuzzing framework like AFL
  - Use valgrind, address-sanitizer, etc.
  - Let your experienced colleagues check your code ☺
- Reuse code when possible and avoid duplication

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
  - Write your own test cases
  - Use static code analysis like *cppcheck* or *scan-build*
  - Use a fuzzing framework like AFL
  - Use valgrind, address-sanitizer, etc.
  - Let your experienced colleagues check your code ☺
- Reuse code when possible and avoid duplication

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
  - Write your own test cases
  - Use static code analysis like *cppcheck* or *scan-build*
  - Use a fuzzing framework like AFL
  - Use valgrind, address-sanitizer, etc.
  - Let your experienced colleagues check your code ☺
- Reuse code when possible and avoid duplication

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
  - Write your own test cases
  - Use static code analysis like *cppcheck* or *scan-build*
  - Use a fuzzing framework like AFL
  - Use valgrind, address-sanitizer, etc.
  - Let your experienced colleagues check your code ☺
- Reuse code when possible and avoid duplication

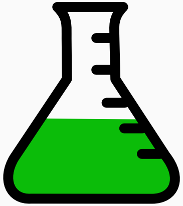
Implementation Flaws II

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

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
  - Write your own test cases
  - Use static code analysis like *cppcheck* or *scan-build*
  - Use a fuzzing framework like AFL
  - Use valgrind, address-sanitizer, etc.
  - Let your experienced colleagues check your code ☺
- Reuse code when possible and avoid duplication

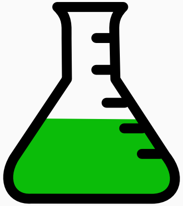
Test cases



- Implement your own exhaustive test cases
- Think of corner cases
  - Invalid ELF header, overlapping ELF sections
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

Overall branch coverage	Bonus points
$65\% <= cov < 70\%$	1
$70\% <= cov < 75\%$	3
$75\% <= cov < 80\%$	5
$80\% <= cov < 85\%$	7
$85\% <= cov < 90\%$	10
$90\% <= cov < 95\%$	15
$95\% <= cov$	20

Test cases



- Implement your own exhaustive test cases
- Think of corner cases
  - Invalid ELF header, overlapping ELF sections
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

Overall branch coverage	Bonus points
$65\% <= cov < 70\%$	1
$70\% <= cov < 75\%$	3
$75\% <= cov < 80\%$	5
$80\% <= cov < 85\%$	7
$85\% <= cov < 90\%$	10
$90\% <= cov < 95\%$	15
$95\% <= cov$	20

Test cases

- Implement your own exhaustive test cases
- Think of corner cases
  - Invalid ELF header, overlapping ELF sections
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

Overall branch coverage	Bonus points
$65\% <= cov < 70\%$	1
$70\% <= cov < 75\%$	3
$75\% <= cov < 80\%$	5
$80\% <= cov < 85\%$	7
$85\% <= cov < 90\%$	10
$90\% <= cov < 95\%$	15
$95\% <= cov$	20

Test cases

- Implement your own exhaustive test cases
- Think of corner cases
  - Invalid ELF header, overlapping ELF sections
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

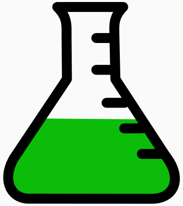
Overall branch coverage	Bonus points
$65\% <= cov < 70\%$	1
$70\% <= cov < 75\%$	3
$75\% <= cov < 80\%$	5
$80\% <= cov < 85\%$	7
$85\% <= cov < 90\%$	10
$90\% <= cov < 95\%$	15
$95\% <= cov$	20

Test cases

- Implement your own exhaustive test cases
- Think of corner cases
  - Invalid ELF header, overlapping ELF sections
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

Overall branch coverage	Bonus points
$65\% <= cov < 70\%$	1
$70\% <= cov < 75\%$	3
$75\% <= cov < 80\%$	5
$80\% <= cov < 85\%$	7
$85\% <= cov < 90\%$	10
$90\% <= cov < 95\%$	15
$95\% <= cov$	20

Test cases



- Implement your own exhaustive test cases
- Think of corner cases
  - Invalid ELF header, overlapping ELF sections
  - NULL pointers, integer overflows, out of mem, ...
- Good coverage yields bonus points (if above 50%)

Overall branch coverage	Bonus points
$65\% <= cov < 70\%$	1
$70\% <= cov < 75\%$	3
$75\% <= cov < 80\%$	5
$80\% <= cov < 85\%$	7
$85\% <= cov < 90\%$	10
$90\% <= cov < 95\%$	15
$95\% <= cov$	20

HELP! How should I start?

- Pull from upstream
- Read the provided `README.md`, `Assignment.md`
- Try to understand basic structure of ELF
- Use `readelf` and examine some binaries
- Nice overview: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
- ELF Segments and Sections `https://lwn.net/Articles/276782/`
- `man elf`
- 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`
- Try to understand basic structure of ELF
- Use `readelf` and examine some binaries
- Nice overview: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
- ELF Segments and Sections `https://lwn.net/Articles/276782/`
- `man elf`
- 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`
- Try to understand basic structure of ELF
- Use `readelf` and examine some binaries
- Nice overview: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
- ELF Segments and Sections `https://lwn.net/Articles/276782/`
- `man elf`
- 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`
- Try to understand basic structure of ELF
- Use `readelf` and examine some binaries
- Nice overview: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
- ELF Segments and Sections `https://lwn.net/Articles/276782/`
- `man elf`
- 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`
- Try to understand basic structure of ELF
- Use `readelf` and examine some binaries
- Nice overview: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
- ELF Segments and Sections `https://lwn.net/Articles/276782/`
- `man elf`
- 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`
- Try to understand basic structure of ELF
- Use `readelf` and examine some binaries
- Nice overview: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
- ELF Segments and Sections `https://lwn.net/Articles/276782/`
- `man elf`
- 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`
- Try to understand basic structure of ELF
- Use `readelf` and examine some binaries
- Nice overview: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
- ELF Segments and Sections `https://lwn.net/Articles/276782/`
- `man elf`
- 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`
- Try to understand basic structure of ELF
- Use **`readelf`** and examine some binaries
- Nice overview: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
- ELF Segments and Sections `https://lwn.net/Articles/276782/`
- `man elf`
- 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`
- Try to understand basic structure of ELF
- Use `readelf` and examine some binaries
- Nice overview: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
- ELF Segments and Sections `https://lwn.net/Articles/276782/`
- `man elf`
- Ask on our Discord channel!
- Come by during question hours!