Secure Software Development

Defensive Programming

Daniel Gruss, Vedad Hadzic, Lukas Maar, Stefan Gast, Marcel Nageler

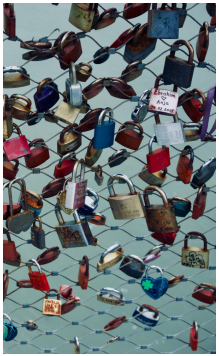
24.11.2023

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

Table of contents

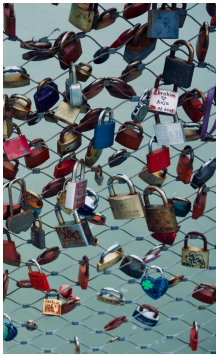
Defense-in-Depth

Defense-in-Depth



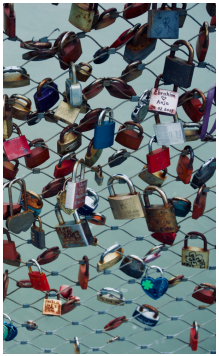
- Understand the attacker's perspective

Defense-in-Depth



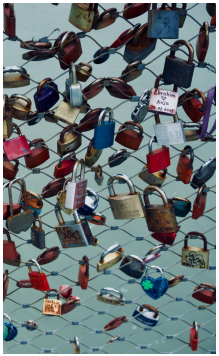
- Understand the attacker's perspective
  - *"Know your enemy"* – Sun Tzu, *The Art of War*

Defense-in-Depth



- Understand the attacker's perspective
  - *"Know your enemy"* – Sun Tzu, *The Art of War*
- Defend on all layers

Defense-in-Depth



- Understand the attacker's perspective
  - *"Know your enemy"* – Sun Tzu, *The Art of War*
- Defend on all layers
  - The weakest link will break first

Defense-In-Depth

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

Defense-In-Depth

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

- Vulnerability prevention (today)

Defense-In-Depth

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

- Vulnerability prevention (today)
- Exploit prevention (next time)

Defense-In-Depth

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

- Vulnerability prevention (today)
- Exploit prevention (next time)
- Privilege minimization (next time)

Defense-In-Depth

Attacker's perspective

- Vulnerability discovery
  - buffer/integer overflow, use-after-free, format strings, type confusion
- Exploitation
  - Data corruption, shellcode, code reuse, ROP, return-to-libc
- Privilege elevation
  - exploit suid binaries, kernel exploits, crack root PW hash ;)

Defense-In-Depth

Attacker's perspective

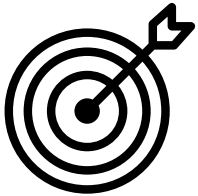
- Vulnerability discovery
  - buffer/integer overflow, use-after-free, format strings, type confusion
- Exploitation
  - Data corruption, shellcode, code reuse, ROP, return-to-libc
- Privilege elevation
  - exploit suid binaries, kernel exploits, crack root PW hash ;)

Defender's perspective

- Vulnerability prevention
  - Code quality, memory safety, type safety, error handling ...
- Exploit prevention
  - Compiler/runtime defenses, hardware defenses
- Privilege minimization
  - System call filtering, sandboxing, virtualization

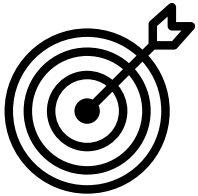
Vulnerability Prevention: Goals I

Vulnerability Prevention: Goals I



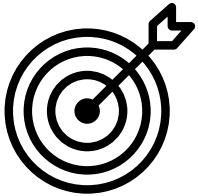
- Overall goal: eliminate **all** vulnerabilities

Vulnerability Prevention: Goals I



- Overall goal: eliminate **all** vulnerabilities
  - Proving a program's correctness is hard

Vulnerability Prevention: Goals I

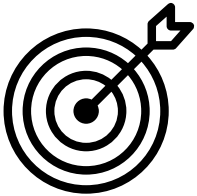


- Overall goal: eliminate **all** vulnerabilities
  - Proving a program's correctness is hard
  - Specification needs to be complete

Vulnerability Prevention: Goals I

- Overall goal: eliminate **all** vulnerabilities
  - Proving a program's correctness is hard
  - Specification needs to be complete
  - Model-based Testing, Verification and Testing

Vulnerability Prevention: Goals II

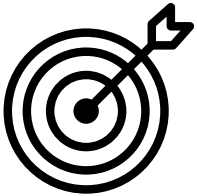


- seL4 formal verification of microkernel took 20.5 person years to verify

Vulnerability Prevention: Goals II

- seL4 formal verification of microkernel took 20.5 person years to verify
- Very limited: less than 10 kloc, no concurrency, ...

Vulnerability Prevention: Goals II



- seL4 formal verification of microkernel took 20.5 person years to verify
- Very limited: less than 10 kloc, no concurrency, ...
- 🏁 Overall goal I: Eliminate as many vulnerabilities as possible (Improve code quality)

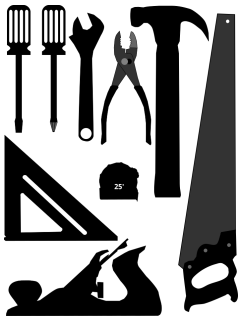
Vulnerability Prevention: Goals II

- seL4 formal verification of microkernel took 20.5 person years to verify
- Very limited: less than 10 kloc, no concurrency, ...

🏁 Overall goal I: Eliminate as many vulnerabilities as possible (Improve code quality)

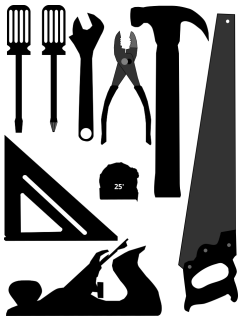
🏁 Overall goal II: Make exploitation as hard as possible (System hardening)

Defensive Programming Overview



Vulnerabilities

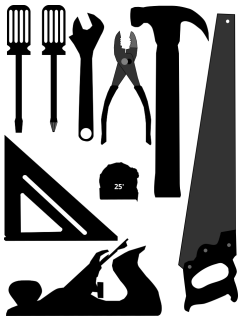
Defensive Programming: Goals & Principles



Vulnerabilities

- Memory safety

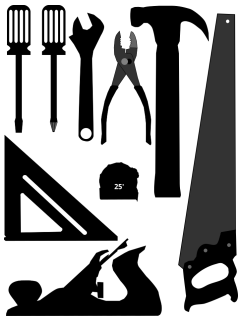
Defensive Programming: Goals & Principles



Vulnerabilities

- Memory safety
- Type safety

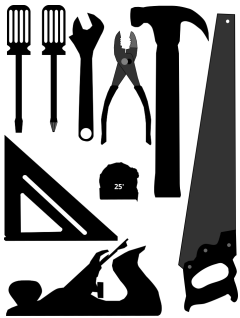
Defensive Programming: Goals & Principles



Vulnerabilities

- Memory safety
- Type safety
- Integer safety

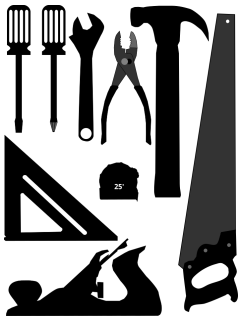
Defensive Programming: Goals & Principles



Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow

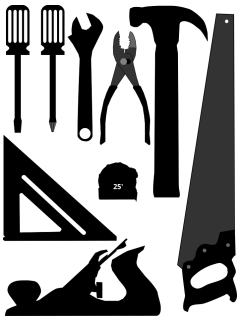
Defensive Programming: Goals & Principles



Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization

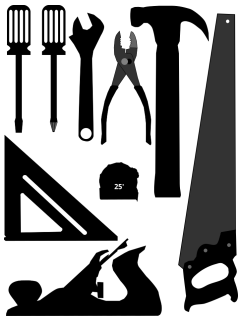
Defensive Programming: Goals & Principles



Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow

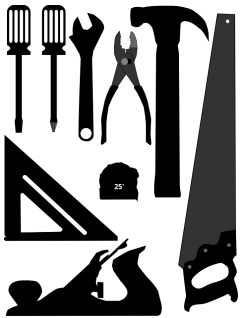
Defensive Programming: Goals & Principles



Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

Defensive Programming: Goals & Principles

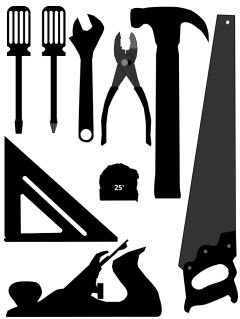


Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

Defensive Programming: Goals & Principles



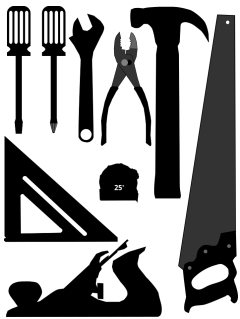
Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language

Defensive Programming: Goals & Principles



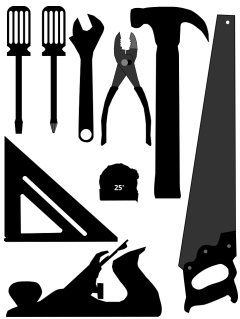
Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:

Defensive Programming: Goals & Principles



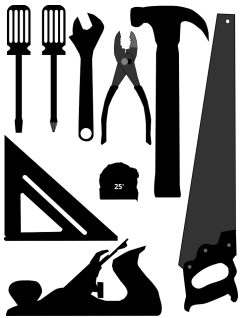
Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard

Defensive Programming: Goals & Principles



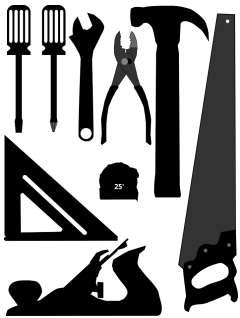
Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse

Defensive Programming: Goals & Principles



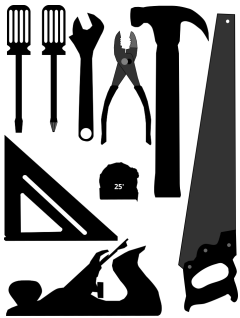
Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions

Defensive Programming: Goals & Principles



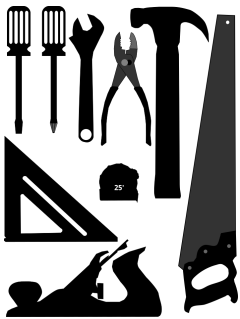
Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation

Defensive Programming: Goals & Principles



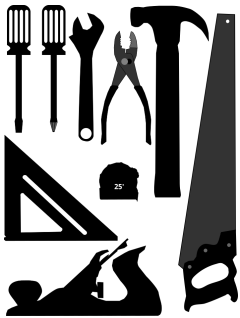
Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions

Defensive Programming: Goals & Principles



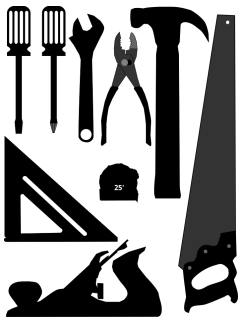
Vulnerabilities

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions
  - Compiler assistance

Defensive Programming: Goals & Principles



Sub-goals

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions
  - Compiler assistance

Spatial memory safety

Spatial memory safety

🏴 Goal: eliminate array out-of-bounds access

Spatial memory safety

- Goal: eliminate array out-of-bounds access
- Use or develop accessor functions which check boundaries

Spatial memory safety

- Goal: eliminate array out-of-bounds access
- Use or develop accessor functions which check boundaries
  - Example: C++ std::vector

Spatial memory safety

- Goal: eliminate array out-of-bounds access
- Use or develop accessor functions which check boundaries
- Example: C++ std::vector

```
int x = vector[index];     // unchecked
int y = vector.at(index); // checked
```

Spatial memory safety

- 🏴 Goal: eliminate array out-of-bounds access
- 👉 Use or develop accessor functions which check boundaries
  - Example: C++ std::vector

```
int x = vector[index];     // unchecked
int y = vector.at(index); // checked
```

  - Memory-safe languages: Compiler can optimize out unnecessary bounds checks

Temporal memory safety

Temporal memory safety

🏁 Goal: eliminate temporal issues

Temporal memory safety

- 🏁 Goal: eliminate temporal issues
- 👉 Use reference counting

Temporal memory safety

- 🏁 Goal: eliminate temporal issues
- 👉 Use reference counting
  - Avoids memory leaks and double free

Temporal memory safety

- 🏁 Goal: eliminate temporal issues
- 👉 Use reference counting
  - Avoids memory leaks and double free
  - E.g., C++ smart pointers `std::shared_ptr` `std::unique_ptr`

Temporal memory safety

- 🏁 Goal: eliminate temporal issues
- 👉 Use reference counting
  - Avoids memory leaks and double free
  - E.g., C++ smart pointers `std::shared_ptr` `std::unique_ptr`
- 👉 Exclusive ownership: only one owner with write access

Temporal memory safety

- 🏁 Goal: eliminate temporal issues
- 👉 Use reference counting
  - Avoids memory leaks and double free
  - E.g., C++ smart pointers `std::shared_ptr` `std::unique_ptr`
- 👉 Exclusive ownership: only one owner with write access
  - Avoids race conditions and TOCTTOU

Temporal memory safety

- 🏁 Goal: eliminate temporal issues
- 👉 Use reference counting
  - Avoids memory leaks and double free
  - E.g., C++ smart pointers `std::shared_ptr` `std::unique_ptr`
- 👉 Exclusive ownership: only one owner with write access
  - Avoids race conditions and TOCTTOU
  - E.g., Rust ownership

Temporal memory safety

- 🏁 Goal: eliminate temporal issues
- 👉 Use reference counting
  - Avoids memory leaks and double free
  - E.g., C++ smart pointers `std::shared_ptr` `std::unique_ptr`
- 👉 Exclusive ownership: only one owner with write access
  - Avoids race conditions and TOCTTOU
  - E.g., Rust ownership
- 👉 Deinitialize free'd resources

Temporal memory safety

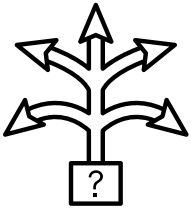
- 🏁 Goal: eliminate temporal issues
- 👉 Use reference counting
  - Avoids memory leaks and double free
  - E.g., C++ smart pointers `std::shared_ptr` `std::unique_ptr`
- 👉 Exclusive ownership: only one owner with write access
  - Avoids race conditions and TOCTTOU
  - E.g., Rust ownership
- 👉 Deinitialize free'd resources
  - Avoids use-after-free (UAF)

Temporal memory safety

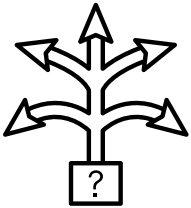
- 🏁 Goal: eliminate temporal issues
- 👉 Use reference counting
  - Avoids memory leaks and double free
  - E.g., C++ smart pointers `std::shared_ptr` `std::unique_ptr`
- 👉 Exclusive ownership: only one owner with write access
  - Avoids race conditions and TOCTTOU
  - E.g., Rust ownership
- 👉 Deinitialize free'd resources
  - Avoids use-after-free (UAF)

```
memset(ptr, 0, size); free(ptr); ptr = NULL;
```

Type safety

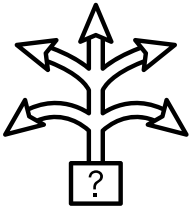


Type safety



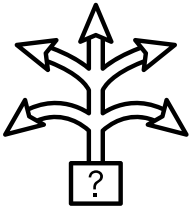
🏴 Goal: eliminate type confusion issues

Type safety



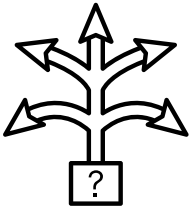
- Goal: eliminate type confusion issues
- Avoid unsafe casts

Type safety



- 🏴 Goal: eliminate type confusion issues
- 👉 Avoid unsafe casts
  - Avoid cast to `(void *)` This should mainly be used for memory allocators

Type safety



- Goal: eliminate type confusion issues
- Avoid unsafe casts
  - Avoid cast to `(void *)` This should mainly be used for memory allocators
  - Avoid C++ `reinterpret_cast`

Fat Pointers

- General concept

Fat Pointers

- General concept
- Store additional runtime metadata alongside pointer, e.g.:

Fat Pointers

- General concept
- Store additional runtime metadata alongside pointer, e.g.:
  - pointer type

Fat Pointers

- General concept
- Store additional runtime metadata alongside pointer, e.g.:
  - pointer type
  - length

Fat Pointers

- General concept
- Store additional runtime metadata alongside pointer, e.g.:
  - pointer type
  - length
  - validity

Fat Pointers

- General concept
- Store additional runtime metadata alongside pointer, e.g.:
  - pointer type
  - length
  - validity
  - access permissions

Fat Pointers

- General concept
- Store additional runtime metadata alongside pointer, e.g.:
  - pointer type
  - length
  - validity
  - access permissions
- Examples: C++ smart pointer

Fat Pointers

- General concept
- Store additional runtime metadata alongside pointer, e.g.:
  - pointer type
  - length
  - validity
  - access permissions
- Examples: C++ smart pointer
- `dynamic_cast<>` is similar but not a fat pointer

Fat Pointers

- General concept
- Store additional runtime metadata alongside pointer, e.g.:
  - pointer type
  - length
  - validity
  - access permissions
- Examples: C++ smart pointer
- `dynamic_cast<>` is similar but not a fat pointer

☹ Typically consume 2–5 times more memory per pointer

Integer Safety I

Integer Safety I

🏴 Goal 1: prevent integer overflows/underflows

Integer Safety I

- Goal 1: prevent integer overflows/underflows
- Use correct integer types

Integer Safety I

- Goal 1: prevent integer overflows/underflows
- Use correct integer types
  - Use `size_t` for indices and length

Integer Safety I

- Goal 1: prevent integer overflows/underflows
- Use correct integer types
  - Use `size_t` for indices and length
  - Use `uint64_t`, etc. for fixed-size integers

Integer Safety I

- Goal 1: prevent integer overflows/underflows
- Use correct integer types
  - Use `size_t` for indices and length
  - Use `uint64_t`, etc. for fixed-size integers
  - Use `uintptr_t` for pointer-to-integer conversion

Integer Safety I

- Goal 1: prevent integer overflows/underflows
- Use correct integer types
  - Use `size_t` for indices and length
  - Use `uint64_t`, etc. for fixed-size integers
  - Use `uintptr_t` for pointer-to-integer conversion
- For **every** arithmetic operation check if overflow is possible

Integer Safety I

- 🏴 Goal 1: prevent integer overflows/underflows
- 👉 Use correct integer types
  - Use `size_t` for indices and length
  - Use `uint64_t`, etc. for fixed-size integers
  - Use `uintptr_t` for pointer-to-integer conversion
- 👉 For **every** arithmetic operation check if overflow is possible
- 👉 Detect and prevent integer overflows via

Integer Safety I

- Goal 1: prevent integer overflows/underflows
- Use correct integer types
  - Use `size_t` for indices and length
  - Use `uint64_t`, etc. for fixed-size integers
  - Use `uintptr_t` for pointer-to-integer conversion
- For **every** arithmetic operation check if overflow is possible
- Detect and prevent integer overflows via
  - Compiler builtins, e.g. `__builtin_saddl_overflow`

Integer Safety I

- Goal 1: prevent integer overflows/underflows
- Use correct integer types
  - Use `size_t` for indices and length
  - Use `uint64_t`, etc. for fixed-size integers
  - Use `uintptr_t` for pointer-to-integer conversion
- For **every** arithmetic operation check if overflow is possible
- Detect and prevent integer overflows via
  - Compiler builtins, e.g. `__builtin_saddl_overflow`
  - Larger types for intermediate results

Integer Safety I

- Goal 1: prevent integer overflows/underflows
- Use correct integer types
  - Use `size_t` for indices and length
  - Use `uint64_t`, etc. for fixed-size integers
  - Use `uintptr_t` for pointer-to-integer conversion
- For **every** arithmetic operation check if overflow is possible
- Detect and prevent integer overflows via
  - Compiler builtins, e.g. `__builtin_saddl_overflow`
  - Larger types for intermediate results
  - Explicit checks

Integer Safety II

🏴 Goal 2: prevent integer conversion issues

Integer Safety II

- Goal 2: prevent integer conversion issues
- Avoid mixing `signed` and `unsigned`

Integer Safety II

- Goal 2: prevent integer conversion issues
- Avoid mixing `signed` and `unsigned`
  - Attention: `char` can be `signed` or `unsigned`

Integer Safety II

- Goal 2: prevent integer conversion issues
- Avoid mixing `signed` and `unsigned`
  - Attention: `char` can be `signed` or `unsigned`
- Make large types **explicit**

Integer Safety II

- Goal 2: prevent integer conversion issues
- Avoid mixing `signed` and `unsigned`
  - Attention: `char` can be `signed` or `unsigned`
- Make large types **explicit**

```
int y = ...;
long long x = y + 2;    // int-addition might overflow
long long x = (long long)y + 2LL;
```

Integer Safety II

- Goal 2: prevent integer conversion issues
- Avoid mixing `signed` and `unsigned`
  - Attention: `char` can be `signed` or `unsigned`
- Make large types **explicit**

```
int y = ...;
long long x = y + 2;    // int-addition might overflow
long long x = (long long)y + 2LL;
```

- Perform range checks on each downcast

Integer Safety II

- Goal 2: prevent integer conversion issues
- Avoid mixing `signed` and `unsigned`
  - Attention: `char` can be `signed` or `unsigned`
- Make large types **explicit**

```
int y = ...;
long long x = y + 2;    // int-addition might overflow
long long x = (long long)y + 2LL;
```

- Perform range checks on each downcast

```
int x = ...;
if (x > SHORT_MAX) error();
short y = (short)x;
```

Integer Safety III

🏴 Goal 3: prevent undefined behavior

Integer Safety III

- Goal 3: prevent undefined behavior
- Examples:

Integer Safety III

- Goal 3: prevent undefined behavior
- Examples:
  - Signed integer overflow

Integer Safety III

- Goal 3: prevent undefined behavior
- Examples:
  - Signed integer overflow
  - Left-shift a signed type

Integer Safety III

- Goal 3: prevent undefined behavior
- Examples:
  - Signed integer overflow
  - Left-shift a signed type
    - number could become negative if MSB is reached

Integer Safety III

- Goal 3: prevent undefined behavior
- Examples:
  - Signed integer overflow
  - Left-shift a signed type
    - number could become negative if MSB is reached
  - Left-shift by a negative number (architecture!)

Integer Safety III

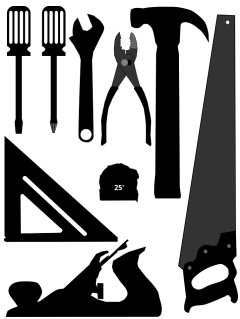
- Goal 3: prevent undefined behavior
- Examples:
  - Signed integer overflow
  - Left-shift a signed type
    - number could become negative if MSB is reached
  - Left-shift by a negative number (architecture!)
  - Unsequenced modifications:

Integer Safety III

- Goal 3: prevent undefined behavior
- Examples:
  - Signed integer overflow
  - Left-shift a signed type
    - number could become negative if MSB is reached
  - Left-shift by a negative number (architecture!)
  - Unsequenced modifications:

```
int f(int i) {
    return i++ + i++; /* undefined behavior: two unsequenced
    modifications to i */
}
```

Defensive Programming: Goals & Principles



Sub-goals

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions
  - Compiler assistance

Secure Data Flow

Secure Data Flow

- Observation: Attacker injects payload as data, which might get misinterpreted as code

Secure Data Flow

- Observation: Attacker injects payload as data, which might get misinterpreted as code
- Idea: Focus on data flow rather than memory objects, types, etc.

Secure Data Flow

- Observation: Attacker injects payload as data, which might get misinterpreted as code
- Idea: Focus on data flow rather than memory objects, types, etc.
- Goal: Secure data flow → attacker cannot inject payload

Secure Data Flow

- Observation: Attacker injects payload as data, which might get misinterpreted as code
- Idea: Focus on data flow rather than memory objects, types, etc.
- Goal: Secure data flow → attacker cannot inject payload
  - Check *every* input an attacker can control directly or indirectly

Secure Data Flow

- 👁 Observation: Attacker injects payload as data, which might get misinterpreted as code
- 💡 Idea: Focus on data flow rather than memory objects, types, etc.
- 🏁 Goal: Secure data flow → attacker cannot inject payload
  - Check *every* input an attacker can control directly or indirectly
  - Better: Check *every* input

Secure Data Flow

- 👁 Observation: Attacker injects payload as data, which might get misinterpreted as code
- 💡 Idea: Focus on data flow rather than memory objects, types, etc.
- 🏴 Goal: Secure data flow → attacker cannot inject payload
  - Check *every* input an attacker can control directly or indirectly
  - Better: Check *every* input
  - 👉 **Input sanitization**

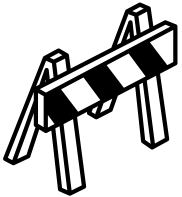
by **Michael Mimoso** Follow @mike_mimoso January 28, 2016 , 9:04 am

Stepankin said he was able to execute arbitrary shell commands on PayPal servers by taking advantage of insecure Java object deserialization. He wrote in a blog post that he was able to access PayPal's production servers.

"I realized that I could execute arbitrary OS commands on manager.paypal.com web servers and moreover, I could establish a back connection to my own internet server and, for example, upload and execute a backdoor," he wrote. "[As a] result, I could get access to production databases used by manager.paypal.com application."

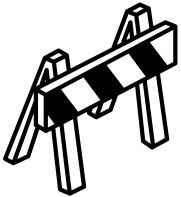
"I just read "/etc/passwd" file by sending it to my server as a proof of the vulnerability," he wrote.

Secure Data Flow: Input Sanitization



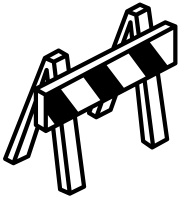
- Goal: sanitize dangerous input

Secure Data Flow: Input Sanitization



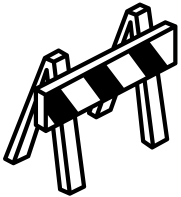
- Goal: sanitize dangerous input
  - Detect and reject

Secure Data Flow: Input Sanitization



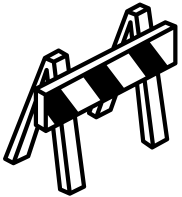
- Goal: sanitize dangerous input
  - Detect and reject
  - Filtering

Secure Data Flow: Input Sanitization



- Goal: sanitize dangerous input
  - Detect and reject
  - Filtering
  - Character escaping

Secure Data Flow: Input Sanitization



- Goal: sanitize dangerous input
  - Detect and reject
  - Filtering
  - Character escaping
  - Use existing sanitizers wherever possible!

Secure Data Flow: Canonicalization I

- Issue: equivalent representations make sanitization a pain

Secure Data Flow: Canonicalization I

Issue: equivalent representations make sanitization a pain

Example: equivalent Unix paths

```
/proc/self/maps
~/../../proc/self/maps
~/../../../../../proc/1/maps
/proc/1/./maps
/proc/1/./////maps
/proc/../proc/1/maps
...
```

Secure Data Flow: Canonicalization II

Idea: make parsing more uniform

Secure Data Flow: Canonicalization II

- Idea: make parsing more uniform
- Canonicalization before Sanitization!

Secure Data Flow: Canonicalization II

- Idea: make parsing more uniform
- Canonicalization before Sanitization!
  - Example: canonicalizing paths using `realpath`

Secure Data Flow: Canonicalization II

- Idea: make parsing more uniform
- Canonicalization before Sanitization!
- Example: canonicalizing paths using `realpath`
  - Resolve relative paths

Secure Data Flow: Canonicalization II

- Idea: make parsing more uniform
- Canonicalization before Sanitization!
- Example: canonicalizing paths using `realpath`
  - Resolve relative paths

```
../../etc/passwd --> /etc/passwd --> DENY
./etc/passwd --> /home/ssd/etc/passwd --> ALLOW
```

Secure Data Flow: Canonicalization II

- Idea: make parsing more uniform
- Canonicalization before Sanitization!
- Example: canonicalizing paths using `realpath`
  - Resolve relative paths

    `../../etc/passwd --> /etc/passwd --> DENY`

    `./etc/passwd --> /home/ssd/etc/passwd --> ALLOW`
  - Resolve symlinks:

Secure Data Flow: Canonicalization II

- Idea: make parsing more uniform
- Canonicalization before Sanitization!
- Example: canonicalizing paths using `realpath`
  - Resolve relative paths

    `../../etc/passwd --> /etc/passwd --> DENY`

    `./etc/passwd --> /home/ssd/etc/passwd --> ALLOW`
  - Resolve symlinks:

    `mylink-to-passwd --> /etc/passwd --> DENY`

Secure Data Flow: Canonicalization III

More issues to consider

- Newlines: `\r\n` vs. `\n`

Secure Data Flow: Canonicalization III

More issues to consider

- Newlines: `\r\n` vs. `\n`
- Path separators: Windows `\` vs. Unix `/`

Secure Data Flow: Canonicalization III

More issues to consider

- Newlines: `\r\n` vs. `\n`
- Path separators: Windows `\` vs. Unix `/`
- Case (in)sensitivity: (Git CVE-2014-9390)

Secure Data Flow: Canonicalization III

More issues to consider

- Newlines: `\r\n` vs. `\n`
- Path separators: Windows \ vs. Unix /
- Case (in)sensitivity: (Git CVE-2014-9390)
- File system permissions: FAT vs NTFS vs EXT

Secure Data Flow: Canonicalization III

More issues to consider

- Newlines: `\r\n` vs. `\n`
- Path separators: Windows `\` vs. Unix `/`
- Case (in)sensitivity: (Git CVE-2014-9390)
- File system permissions: FAT vs NTFS vs EXT
- Time stamps (UTC vs GMT)

Secure Data Flow: Character Escaping I

- Issue: user input gets interpreted as code, e.g.:

Secure Data Flow: Character Escaping I

- Issue: user input gets interpreted as code, e.g.:
  - SQL injection

Secure Data Flow: Character Escaping I

- Issue: user input gets interpreted as code, e.g.:
  - SQL injection
  - Cross-site scripting (HTML injection)

Secure Data Flow: Character Escaping I

- Issue: user input gets interpreted as code, e.g.:
  - SQL injection
  - Cross-site scripting (HTML injection)
  - Shell command injection

Secure Data Flow: Character Escaping I

Issue: user input gets interpreted as code, e.g.:

- SQL injection
- Cross-site scripting (HTML injection)
- Shell command injection

```
Please enter your username: Dr.`whoami`
User Dr.Gruss created.
```

Secure Data Flow: Character Escaping I

Issue: user input gets interpreted as code, e.g.:

- SQL injection
- Cross-site scripting (HTML injection)
- Shell command injection

```
Please enter your username: Dr.`whoami`
User Dr.Gruss created.
```

Never interpret user input as code!

Secure Data Flow: Character Escaping I

- Issue: user input gets interpreted as code, e.g.:
  - SQL injection
  - Cross-site scripting (HTML injection)
  - Shell command injection

```
Please enter your username: Dr.`whoami`
User Dr.Gruss created.
```

- Never interpret user input as code!
- If not possible: neutralize dangerous characters

Secure Data Flow: Character Escaping I

- Issue: user input gets interpreted as code, e.g.:
  - SQL injection
  - Cross-site scripting (HTML injection)
  - Shell command injection

```
Please enter your username: Dr.`whoami`
User Dr.Gruss created.
```

- Never interpret user input as code!
- If not possible: neutralize dangerous characters
- Escaping before Interpretation!

Secure Data Flow: Character Escaping II

- Examples:

Secure Data Flow: Character Escaping II

- Examples:
  - Shell: replace ` with \`

Secure Data Flow: Character Escaping II

- Examples:
  - Shell: replace `` ` `` with `` \` ``
  - HTML: replace `<script>` with `<script>`

Secure Data Flow: Character Escaping II

- Examples:
  - Shell: replace `` ` `` with `` \` ``
  - HTML: replace `<script>` with `<script>`

Do not reinvent escaping!

Secure Data Flow: Character Escaping II

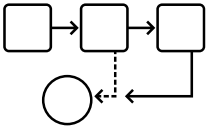
- Examples:
  - Shell: replace `` ` `` with `` \` ``
  - HTML: replace `<script>` with `<script>`

Do not reinvent escaping!

Use existing libraries

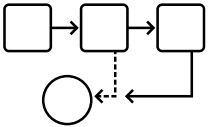
Secure Control Flow

Secure Control Flow: Proper Error Handling



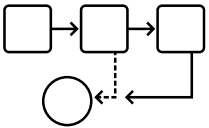
- Return error codes

Secure Control Flow: Proper Error Handling



- Return error codes
- Exception handling

Secure Control Flow: Proper Error Handling



- Return error codes
- Exception handling
- Goto

Secure Control Flow: Return Error Codes I

```
1  FILE* f = fopen("report.log", "a");
2  fprintf(f, "Server started\n");
3  printf("DEBUG: we're running\n");
4  fclose(f);
```

Secure Control Flow: Return Error Codes I

```
1  FILE* f = fopen("report.log", "a");
2  fprintf(f, "Server started\n");
3  printf("DEBUG: we're running\n");
4  fclose(f);
```

- Is this code clean?

Secure Control Flow: Return Error Codes I

```
1  FILE* f = fopen("report.log", "a");
2  fprintf(f, "Server started\n");
3  printf("DEBUG: we're running\n");
4  fclose(f);
```

- Is this code clean?

👉 Always check for error codes (line 1 & 2)

Secure Control Flow: Return Error Codes I

```
1  FILE* f = fopen("report.log", "a");
2  fprintf(f, "Server started\n");
3  printf("DEBUG: we're running\n");
4  fclose(f);
```

- Is this code clean?

👉 Always check for error codes (line 1 & 2)

- Exceptions

Secure Control Flow: Return Error Codes I

```
1  FILE* f = fopen("report.log", "a");
2  fprintf(f, "Server started\n");
3  printf("DEBUG: we're running\n");
4  fclose(f);
```

- Is this code clean?

☞ Always check for error codes (line 1 & 2)

- Exceptions
  - If there's nothing you can do in case of an error

Secure Control Flow: Return Error Codes I

```
1  FILE* f = fopen("report.log", "a");
2  fprintf(f, "Server started\n");
3  printf("DEBUG: we're running\n");
4  fclose(f);
```

- Is this code clean?

👉 Always check for error codes (line 1 & 2)

- Exceptions
  - If there's nothing you can do in case of an error
  - In particular cleanup routines `fclose`, `munmap`, (`free`)

Secure Control Flow: Return Error Codes II

```
1  FILE* f = fopen("report.log", "a");
2  if (NULL == f) { perror("Unable to open file"); return; }
3  assert(0 > fprintf(f, "Server started\n"));
4  printf("DEBUG: we're running\n");
5  fclose(f);
```

Secure Control Flow: Return Error Codes II

```
1  FILE* f = fopen("report.log", "a");
2  if (NULL == f) { perror("Unable to open file"); return; }
3  assert(0 > fprintf(f, "Server started\n"));
4  printf("DEBUG: we're running\n");
5  fclose(f);
```

- Is it better now?

Secure Control Flow: Return Error Codes II

```
1  FILE* f = fopen("report.log", "a");
2  if (NULL == f) { perror("Unable to open file"); return; }
3  assert(0 > fprintf(f, "Server started\n"));
4  printf("DEBUG: we're running\n");
5  fclose(f);
```

- Is it better now?

Compiler might optimize out `asserts` (line 3)

Secure Control Flow: Return Error Codes II

```
1  FILE* f = fopen("report.log", "a");
2  if (NULL == f) { perror("Unable to open file"); return; }
3  assert(0 > fprintf(f, "Server started\n"));
4  printf("DEBUG: we're running\n");
5  fclose(f);
```

- Is it better now?

Compiler might optimize out `asserts` (line 3)

- `fprintf` is never executed!

Secure Control Flow: Return Error Codes II

```
1  FILE* f = fopen("report.log", "a");
2  if (NULL == f) { perror("Unable to open file"); return; }
3  assert(0 > fprintf(f, "Server started\n"));
4  printf("DEBUG: we're running\n");
5  fclose(f);
```

- Is it better now?

- Compiler might optimize out `asserts` (line 3)
  - `fprintf` is never executed!
  - Compile flag `-DNDEBUG`

Secure Control Flow: Return Error Codes II

```
1  FILE* f = fopen("report.log", "a");
2  if (NULL == f) { perror("Unable to open file"); return; }
3  assert(0 > fprintf(f, "Server started\n"));
4  printf("DEBUG: we're running\n");
5  fclose(f);
```

- Is it better now?

Compiler might optimize out `asserts` (line 3)

- `fprintf` is never executed!
- Compile flag `-DNDEBUG`

Never use `assert` to check for actual error codes

Secure Control Flow: Return Error Codes III

```
1  char* tmp = realloc(buffer, newsize);
2  if (NULL == tmp) { return REALLOC_FAILED; }
```

Secure Control Flow: Return Error Codes III

```
1  char* tmp = realloc(buffer, newsize);
2  if (NULL == tmp) { return REALLOC_FAILED; }
```

What if `newsize` is 0?

Secure Control Flow: Return Error Codes III

```
1  char* tmp = realloc(buffer, newsize);
2  if (NULL == tmp) { return REALLOC_FAILED; }
```

What if `newsize` is 0?

> The realloc() function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type and may be different from ptr, or NULL if the request fails. If size was equal to 0, either NULL or a pointer suitable to be passed to free() is returned. If realloc() fails, the original block is left untouched; it is not freed or moved.

Secure Control Flow: Return Error Codes III

```
1  char* tmp = realloc(buffer, newsize);
2  if (NULL == tmp) { return REALLOC_FAILED; }
```

What if `newsize` is 0?

> The realloc() function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type and may be different from ptr, or NULL if the request fails. If size was equal to 0, either NULL or a pointer suitable to be passed to free() is returned. If realloc() fails, the original block is left untouched; it is not freed or moved.

👉 Consider all possible error combinations

Secure Control Flow: Return Error Codes III

```
1  char* tmp = realloc(buffer, newsize);
2  if (NULL == tmp) { return REALLOC_FAILED; }
```

What if `newsize` is 0?

> The realloc() function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type and may be different from ptr, or NULL if the request fails. If size was equal to 0, either NULL or a pointer suitable to be passed to free() is returned. If realloc() fails, the original block is left untouched; it is not freed or moved.

👉 Consider all possible error combinations

```
1  char* tmp = realloc(buffer, newsize);
2  if (NULL == tmp && newsize > 0) { return REALLOC_FAILED; }
```

Secure Control Flow: Return Error Codes III

```
1  char* tmp = realloc(buffer, newsize);
2  if (NULL == tmp) { return REALLOC_FAILED; }
```

What if `newsize` is 0?

> The realloc() function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type and may be different from ptr, or NULL if the request fails. If size was equal to 0, either NULL or a pointer suitable to be passed to free() is returned. If realloc() fails, the original block is left untouched; it is not freed or moved.

- Consider all possible error combinations

```
1  char* tmp = realloc(buffer, newsize);
2  if (NULL == tmp && newsize > 0) { return REALLOC_FAILED; }
```

- Make sure your own functions return proper error codes

Secure Control Flow: Return Error Codes IV

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>
FILE* f = fopen("foo","rb");
if(f == NULL) {
  //prints:
  //Error Number: 2 No such file or directory
  fprintf(stderr,"Error Number: %d %s", errno, strerror(errno));
  exit(errno);
}
```


EXCEPTIONS
EXCEPTIONS EVERYWHERE
makeameme.org

Secure Control Flow: Exceptions II

- Exceptions can make your code clean

Image source: `https://www.viva64.com/en/b/0426/`

Secure Control Flow: Exceptions II

- Exceptions can make your code clean
- Exceptions can make your code fast

Image source: `https://www.viva64.com/en/b/0426/`

Secure Control Flow: Exceptions II

- Exceptions can make your code clean
- Exceptions can make your code fast
- Exceptions can make your code expensive

Image source: `https://www.viva64.com/en/b/0426/`

Secure Control Flow: Exceptions II

- Exceptions can make your code clean
- Exceptions can make your code fast
- Exceptions can make your code expensive
- $500 million for a crashed Ariane5 rocket in 1996

Image source: `https://www.viva64.com/en/b/0426/`

Secure Control Flow: Exceptions II

- Exceptions can make your code clean
- Exceptions can make your code fast
- Exceptions can make your code expensive
- $500 million for a crashed Ariane5 rocket in 1996
  - *"The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value."*

Image source: `https://www.viva64.com/en/b/0426/`

Secure Control Flow: Exceptions II

- Exceptions can make your code clean
- Exceptions can make your code fast
- Exceptions can make your code expensive
- $500 million for a crashed Ariane5 rocket in 1996
  - *"The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value."*

`http://sunnyday.mit.edu/nasa-class/Ariane5-report.html`

Image source: `https://www.viva64.com/en/b/0426/`

Secure Control Flow: C++ Exceptions I

- Issue: Exceptions can be hard to comprehend

Secure Control Flow: C++ Exceptions I

- Issue: Exceptions can be hard to comprehend
  - Which statement can throw which exception?

Secure Control Flow: C++ Exceptions I

- Issue: Exceptions can be hard to comprehend
  - Which statement can throw which exception?

```
try
{
  MyClass* a = new MyClass();              // std::bad_alloc exception
  MyClass& b = dynamic_cast<MyClass&>(c); // std::bad_cast exception
}
catch (std::exception& e)
{
  std::cout << "Exception: " << e.what() << std::endl;
}
```

Secure Control Flow: C++ Exceptions I

- Issue: Exceptions can be hard to comprehend
  - Which statement can throw which exception?

```
try
{
  MyClass* a = new MyClass();              // std::bad_alloc exception
  MyClass& b = dynamic_cast<MyClass&>(c); // std::bad_cast exception
}
catch (std::exception& e)
{
  std::cout << "Exception: " << e.what() << std::endl;
}
```

- Exceptions do not save you from thinking through all possibilities

Secure Control Flow: C++ Exceptions II

- Only use exceptions for error cases

Secure Control Flow: C++ Exceptions II

- Only use exceptions for error cases
- Specify which exceptions your function throws

Secure Control Flow: C++ Exceptions II

- Only use exceptions for error cases
- Specify which exceptions your function throws
  - This must include exceptions your function does not catch

Secure Control Flow: C++ Exceptions II

- Only use exceptions for error cases
- Specify which exceptions your function throws
  - This must include exceptions your function does not catch
- Catch exceptions at the correct location

Secure Control Flow: C++ Exceptions II

- Only use exceptions for error cases
- Specify which exceptions your function throws
  - This must include exceptions your function does not catch
- Catch exceptions at the correct location
  - `main` is likely the wrong location

Secure Control Flow: C++ Exceptions II

- Only use exceptions for error cases
- Specify which exceptions your function throws
  - This must include exceptions your function does not catch
- Catch exceptions at the correct location
  - `main` is likely the wrong location
- Watch out for information leakage if your webserver publicly dumps the exception trace ;)

Secure Control Flow: C goto for the Rescue I

- Idea: use `goto` as a C-replacement for exceptions

Secure Control Flow: C goto for the Rescue I

- Idea: use `goto` as a C-replacement for exceptions
  - Write cleanup error code only once

Secure Control Flow: C goto for the Rescue I

- Idea: use `goto` as a C-replacement for exceptions
  - Write cleanup error code only once
- Use `goto` for nothing else

Secure Control Flow: C goto for the Rescue I

- Idea: use `goto` as a C-replacement for exceptions
  - Write cleanup error code only once
- Use `goto` for nothing else
  - Only jump forward, not backward

Secure Control Flow: C goto for the Rescue II

```c
char *resourceA = NULL;
FILE *resourceB = NULL;
void *resourceC = MAP_FAILED;
int err = SUCCESS;

resourceA = malloc(10);
if (NULL == resourceA) {
    err = ERROR_A; goto failed;
}
resourceB = fopen(...);
if (NULL == resourceB) {
    err = ERROR_B; goto failed;
}
resourceC = mmap(...);
if (MAP_FAILED == resourceC) {
    err = ERROR_C; goto failed;
}
return SUCCESS;
```

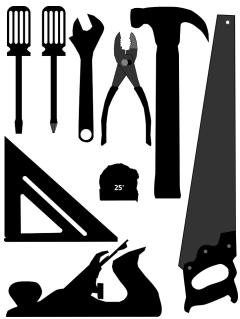
Secure Control Flow: C goto for the Rescue II

```c
char *resourceA = NULL;
FILE *resourceB = NULL;
void *resourceC = MAP_FAILED;
int err = SUCCESS;

resourceA = malloc(10);
if (NULL == resourceA) {
    err = ERROR_A; goto failed;
}
resourceB = fopen(...);
if (NULL == resourceB) {
    err = ERROR_B; goto failed;
}
resourceC = mmap(...);
if (MAP_FAILED == resourceC) {
    err = ERROR_C; goto failed;
}
return SUCCESS;
```

```c
failed:
    if (MAP_FAILED != resourceC) {
        munmap(resourceC);
    }
    if (NULL != resourceB) {
        fclose(resourceB);
    }
    free(resourceA);
    return err;
```

Defensive Programming: Goals & Principles



Sub-goals

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions
  - Compiler assistance

Defensive Programming: Choose Appropriate Language

- Important characteristics

Defensive Programming: Choose Appropriate Language

- Important characteristics
  - Performance

Defensive Programming: Choose Appropriate Language

- Important characteristics
  - Performance
  - Security/safety

Defensive Programming: Choose Appropriate Language

- Important characteristics
  - Performance
  - Security/safety
  - Features

Defensive Programming: Choose Appropriate Language

- Important characteristics
  - Performance
  - Security/safety
  - Features
  - Maintainability

Defensive Programming: Choose Appropriate Language

- Important characteristics
  - Performance
  - Security/safety
  - Features
  - Maintainability
  - Support / community

Defensive Programming: Choose Appropriate Language

- Important characteristics
  - Performance
  - Security/safety
  - Features
  - Maintainability
  - Support / community
  - Portability

Defensive Programming: Choose Appropriate Language

- Important characteristics
  - Performance
  - Security/safety
  - Features
  - Maintainability
  - Support / community
  - Portability
  - ...

Defensive Programming: Choose Appropriate Language

- C, C++

Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance

Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance
  - Inherently unsafe in many aspects

Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance
  - Inherently unsafe in many aspects
  - Important use cases

Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance
  - Inherently unsafe in many aspects
  - Important use cases
    - C: Low-level kernel/embedded development

Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance
  - Inherently unsafe in many aspects
  - Important use cases
    - C: Low-level kernel/embedded development
    - C++: High-performance application development

Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance
  - Inherently unsafe in many aspects
  - Important use cases
    - C: Low-level kernel/embedded development
    - C++: High-performance application development
    - Maintenance of legacy code

Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance
  - Inherently unsafe in many aspects
  - Important use cases
    - C: Low-level kernel/embedded development
    - C++: High-performance application development
    - Maintenance of legacy code
    - Education

Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance
  - Inherently unsafe in many aspects
  - Important use cases
    - C: Low-level kernel/embedded development
    - C++: High-performance application development
    - Maintenance of legacy code
    - Education
- Rust/Carbon

Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance
  - Inherently unsafe in many aspects
  - Important use cases
    - C: Low-level kernel/embedded development
    - C++: High-performance application development
    - Maintenance of legacy code
    - Education
- Rust/Carbon
  - High performance

Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance
  - Inherently unsafe in many aspects
  - Important use cases
    - C: Low-level kernel/embedded development
    - C++: High-performance application development
    - Maintenance of legacy code
    - Education
- Rust/Carbon
  - High performance
  - Memory safety and type safety by design

Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance
  - Inherently unsafe in many aspects
  - Important use cases
    - C: Low-level kernel/embedded development
    - C++: High-performance application development
    - Maintenance of legacy code
    - Education
- Rust/Carbon
  - High performance
  - Memory safety and type safety by design
  - Except `unsafe` keyword

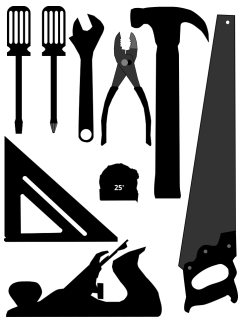
Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance
  - Inherently unsafe in many aspects
  - Important use cases
    - C: Low-level kernel/embedded development
    - C++: High-performance application development
    - Maintenance of legacy code
    - Education
- Rust/Carbon
  - High performance
  - Memory safety and type safety by design
  - Except `unsafe` keyword
  - Main competitor of C

Defensive Programming: Choose Appropriate Language

- C, C++
  - High control, high performance
  - Inherently unsafe in many aspects
  - Important use cases
    - C: Low-level kernel/embedded development
    - C++: High-performance application development
    - Maintenance of legacy code
    - Education
- Rust/Carbon
  - High performance
  - Memory safety and type safety by design
  - Except `unsafe` keyword
  - Main competitor of C
- App / Web development: look out for other languages ...

Defensive Programming: Goals & Principles



Sub-goals

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions
  - Compiler assistance

Improve Code Quality

- Idea: By improving code quality we

Improve Code Quality

- Idea: By improving code quality we
  - Make code more comprehensible to us and to outsiders

Improve Code Quality

- Idea: By improving code quality we
  - Make code more comprehensible to us and to outsiders
  - Decrease likelihood of introducing bugs

Improve Code Quality

- Idea: By improving code quality we
  - Make code more comprehensible to us and to outsiders
  - Decrease likelihood of introducing bugs
  - Increase likelihood of finding bugs

Improve Code Quality

Idea: By improving code quality we

- Make code more comprehensible to us and to outsiders
- Decrease likelihood of introducing bugs
- Increase likelihood of finding bugs
- Make code maintainable

Improve Code Quality

- Idea: By improving code quality we
  - Make code more comprehensible to us and to outsiders
  - Decrease likelihood of introducing bugs
  - Increase likelihood of finding bugs
  - Make code maintainable
  - Make code reusable

Improve Code Quality: Coding Standard I

- Use descriptive and consistent naming scheme,
  e.g. `local_var`, `myFunc`, `_internalFunc`, `MACRO`, `CONSTANT`

Improve Code Quality: Coding Standard I

- Use descriptive and consistent naming scheme,
  e.g. `local_var`, `myFunc`, `_internalFunc`, `MACRO`, `CONSTANT`
- Wrap if/else/for bodies into { ... }

Improve Code Quality: Coding Standard I

- Use descriptive and consistent naming scheme,
  e.g. `local_var`, `myFunc`, `_internalFunc`, `MACRO`, `CONSTANT`
- Wrap if/else/for bodies into { ... }
- Check/sanitize function parameters and return values

Improve Code Quality: Coding Standard I

- Use descriptive and consistent naming scheme,
  e.g. `local_var`, `myFunc`, `_internalFunc`, `MACRO`, `CONSTANT`
- Wrap if/else/for bodies into { ... }
- Check/sanitize function parameters and return values

```
int readPassword(const char* path_password) {
    if (NULL == path_password) {
        return ERROR;
    }
    FILE* file_password = fopen(path_password, "r");
    if (NULL != file_password) {
        return ERROR;
    }
    ...
```

Improve Code Quality: Coding Standard II

- Use C macros with special care

Improve Code Quality: Coding Standard II

- Use C macros with special care
  - Wrap multiple statements in `do { ... } while(0)`

Improve Code Quality: Coding Standard II

- Use C macros with special care
  - Wrap multiple statements in `do { ... } while(0)`
  - Wrap overall macro expression and each argument in `(...)`

Improve Code Quality: Coding Standard II

- Use C macros with special care
  - Wrap multiple statements in `do { ... } while(0)`
  - Wrap overall macro expression and each argument in `(...)`
  - Copy macro parameters if used multiple times. It might be a statement with side effects e.g., `i++`

Improve Code Quality: Coding Standard II

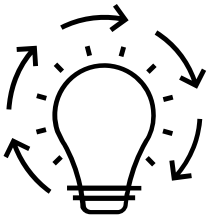
- Use C macros with special care
  - Wrap multiple statements in `do { ... } while(0)`
  - Wrap overall macro expression and each argument in `(...)`
  - Copy macro parameters if used multiple times. It might be a statement with side effects e.g., `i++`
  - If macro has control-flow statements (`return`, `break`, `goto`), include them in the macro name

Improve Code Quality: Coding Standard II

- Use C macros with special care
  - Wrap multiple statements in `do { ... } while(0)`
  - Wrap overall macro expression and each argument in `(...)`
  - Copy macro parameters if used multiple times. It might be a statement with side effects e.g., `i++`
  - If macro has control-flow statements (`return`, `break`, `goto`), include them in the macro name

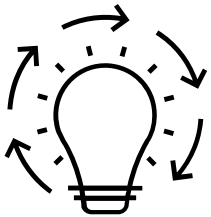
```
#define CHECK_RETURN_ON_ERROR(stmt, err_msg) do { \
  int result = (int)(stmt); \
  if (result < 0) { \
      printf("CHECK failed with %d: %s", result, (err_msg)); \
      return result; \
  } \
} while(0)
```

Improve Code Quality: Intelligent Source Code Reuse



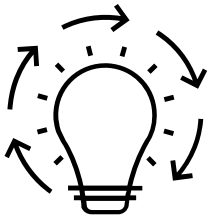
- “Do not rewrite your own $THING”, especially

Improve Code Quality: Intelligent Source Code Reuse



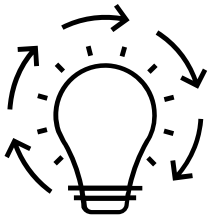
- "Do not rewrite your own $THING", especially
  - Parsers

Improve Code Quality: Intelligent Source Code Reuse



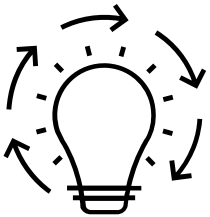
- "Do not rewrite your own $THING", especially
  - Parsers
  - Sanitizers

Improve Code Quality: Intelligent Source Code Reuse



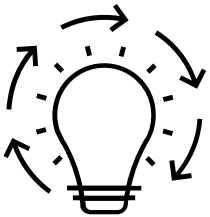
- "Do not rewrite your own $THING", especially
  - Parsers
  - Sanitizers
  - Crypto libraries

Improve Code Quality: Intelligent Source Code Reuse



- ✋ "Do not rewrite your own $THING", especially
  - Parsers
  - Sanitizers
  - Crypto libraries
- 👉 Reuse **established** libraries

Improve Code Quality: Intelligent Source Code Reuse



- ✋ "Do not rewrite your own $THING", especially
  - Parsers
  - Sanitizers
  - Crypto libraries
- 👉 Reuse **established** libraries
  - Stackoverflow/Copilot should be taken with care

PENTEST .BLOG

ADVISORIES OPERATING SYSTEM APPLICATION SECURITY NETWORK TOOLS

One ring to rule them all – Same RCE on multiple Trend Micro products

October 8, 2017 Mehmet Ince Research

One ~~ring~~ bug to rule them all – Widgets of Trend Micro's Products

Most of the Trend Micro's products have a widgets for administrator web page. Although core system written with Java/.NET, this widget mechanism had implemented with PHP. That means, they somehow need to put PHP interpreter on product whenever they decided to use widgets. Which makes it a perfect spot to what we need: a single code base, exist across the different product and awesome way to implement reliable exploit once we have an vulnerability.

For the reasons that I've mentioned above, I performed a code audit for widget system of **Trend Micro OfficeScan** product. Result is quite interesting as well as unfortunate for me. I've found 6 different vulnerability but only 2 of them is **0day**.

[…]

Conclusion

First of all, I would like to say again, this command injection vulnerability has been patched by Trend Micro for both of these products. If you are a Trend Micro user or your organisation is using any of these products, hurry up! Patch your system.

Having same code base on different products is not something bad. I just wanted to point out that one bug within your framework can cause a massive trouble.

Assume that your code will be reused anywhere

Assume that your code will be reused anywhere

- E.g., Ariane 5 reused code from old rocket without checking its assumptions

Improve Code Quality: Assumptions I

- Idea 1: make no assumptions about undefined or implementation-defined behavior

Improve Code Quality: Assumptions I

- Idea 1: make no assumptions about undefined or implementation-defined behavior
  - Signed overflows

Improve Code Quality: Assumptions I

- Idea 1: make no assumptions about undefined or implementation-defined behavior
  - Signed overflows
  - Binary shift operation with a negative shift value

Improve Code Quality: Assumptions I

- Idea 1: make no assumptions about undefined or implementation-defined behavior
  - Signed overflows
  - Binary shift operation with a negative shift value
  - Return value of {m,c,re}alloc if size is zero (implementation-defined)

Improve Code Quality: Assumptions I

- Idea 1: make no assumptions about undefined or implementation-defined behavior
  - Signed overflows
  - Binary shift operation with a negative shift value
  - Return value of {m,c,re}alloc if size is zero (implementation-defined)
- Know undefined behavior of your programming language

  `https://wiki.sei.cmu.edu/confluence/display/c/CC.+Undefined+Behavior`

  See no. 51, 52, 53...

Improve Code Quality: Assumptions II

- Not all code is fully portable

Improve Code Quality: Assumptions II

- Not all code is fully portable

Hidden assumptions are dangerous, e.g., `sizeof(int)`

Improve Code Quality: Assumptions II

- Not all code is fully portable
- Hidden assumptions are dangerous, e.g., `sizeof(int)`
- Idea 2: make all hidden assumptions explicit

Improve Code Quality: Assumptions II

- Not all code is fully portable
- Hidden assumptions are dangerous, e.g., `sizeof(int)`
- Idea 2: make all hidden assumptions explicit
- Document them in the comments

Improve Code Quality: Assumptions II

- Not all code is fully portable
- Hidden assumptions are dangerous, e.g., `sizeof(int)`
- Idea 2: make all hidden assumptions explicit
- Document them in the comments
- Document them in the code using static asserts and `#ifdef...#error`

Improve Code Quality: Assumptions II

- Not all code is fully portable
- Hidden assumptions are dangerous, e.g., `sizeof(int)`
- Idea 2: make all hidden assumptions explicit
- Document them in the comments
- Document them in the code using static asserts and `#ifdef...#error`
  - Both make the compiler fail, thus prevent misuse

Improve Code Quality: Assumptions II

- Not all code is fully portable
- Hidden assumptions are dangerous, e.g., `sizeof(int)`
- Idea 2: make all hidden assumptions explicit
- Document them in the comments
- Document them in the code using static asserts and `#ifdef...#error`
  - Both make the compiler fail, thus prevent misuse
  - ☹ Old C does not support static asserts → hacky macros

    `http://www.pixelbeat.org/programming/gcc/static_assert.html`

Improve Code Quality: Assumptions II

- Not all code is fully portable
- Hidden assumptions are dangerous, e.g., `sizeof(int)`
- Idea 2: make all hidden assumptions explicit
- Document them in the comments
- Document them in the code using static asserts and `#ifdef...#error`
  - Both make the compiler fail, thus prevent misuse
  - ☹ Old C does not support static asserts → hacky macros

    `http://www.pixelbeat.org/programming/gcc/static_assert.html`
  - ☺ Finally, C11 has native support for static asserts

    ```
    static_assert(sizeof(int) == 4, "int must be 4 bytes");
    ```

Improve Code Quality: Documentation

- Write documentation

Improve Code Quality: Documentation

- Write documentation
  - Functional behavior

Improve Code Quality: Documentation

- Write documentation
  - Functional behavior
  - Assumptions about input parameters (e.g. overlapping buffers in `memcpy`/`memmove`)

Improve Code Quality: Documentation

- Write documentation
  - Functional behavior
  - Assumptions about input parameters (e.g. overlapping buffers in `memcpy`/`memmove`)
  - Assumptions about architecture, stdlib version, etc.

Improve Code Quality: Documentation

- Write documentation
  - Functional behavior
  - Assumptions about input parameters (e.g. overlapping buffers in `memcpy`/`memmove`)
  - Assumptions about architecture, stdlib version, etc.
  - Error behavior

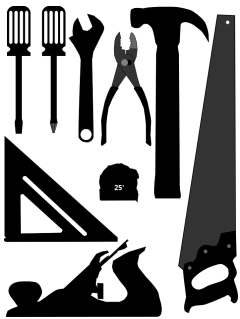
Improve Code Quality: Documentation

- Write documentation
  - Functional behavior
  - Assumptions about input parameters (e.g. overlapping buffers in `memcpy`/`memmove`)
  - Assumptions about architecture, stdlib version, etc.
  - Error behavior
  - Globally visible side-effects (global and static variables)

Improve Code Quality: Documentation

- Write documentation
  - Functional behavior
  - Assumptions about input parameters (e.g. overlapping buffers in `memcpy`/`memmove`)
  - Assumptions about architecture, stdlib version, etc.
  - Error behavior
  - Globally visible side-effects (global and static variables)
  - Multithreading safety

Defensive Programming: Goals & Principles



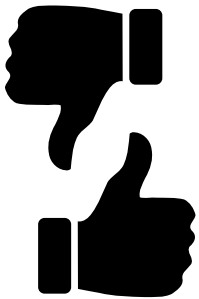
Sub-goals

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

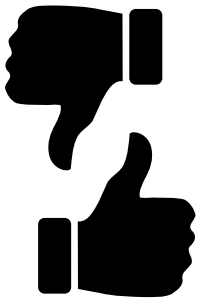
- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions
  - Compiler assistance

Improve Code Quality: Testing & Assertions I



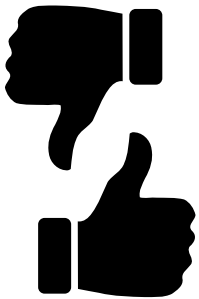
- Testing covered in other courses

Improve Code Quality: Testing & Assertions I



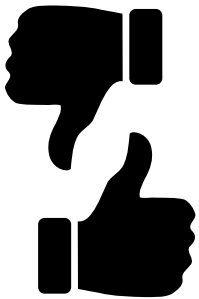
- Testing covered in other courses
- When to use asserts and when not?

Improve Code Quality: Testing & Assertions I



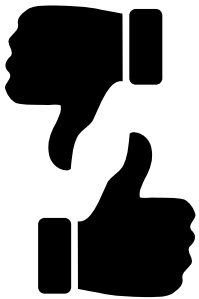
- Testing covered in other courses
- When to use asserts and when not?
- Idea: Asserts reflect invariants of a program

Improve Code Quality: Testing & Assertions I



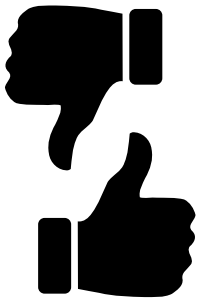
- Testing covered in other courses
- When to use asserts and when not?
- Idea: Asserts reflect invariants of a program
  - In a correct program the invariants would always be satisfied

Improve Code Quality: Testing & Assertions I



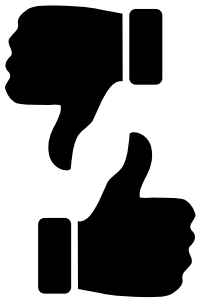
- Testing covered in other courses
- When to use asserts and when not?
- Idea: Asserts reflect invariants of a program
  - In a correct program the invariants would always be satisfied
  - A violation triggers an immediate program abort

Improve Code Quality: Testing & Assertions I



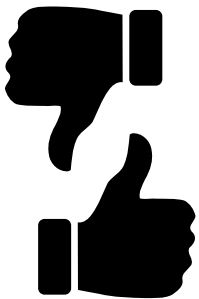
- Testing covered in other courses
- When to use asserts and when not?
- 💡 Idea: Asserts reflect invariants of a program
  - In a correct program the invariants would always be satisfied
  - A violation triggers an immediate program abort
- 👉 Asserts are good for testing

Improve Code Quality: Testing & Assertions I



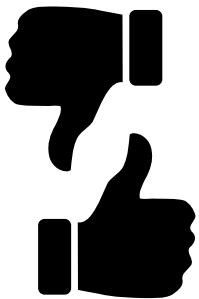
- Testing covered in other courses
- When to use asserts and when not?
- 💡 Idea: Asserts reflect invariants of a program
  - In a correct program the invariants would always be satisfied
  - A violation triggers an immediate program abort
- 👍 Asserts are good for testing
- 👍 Asserts are typically turned off for release build

Improve Code Quality: Testing & Assertions I



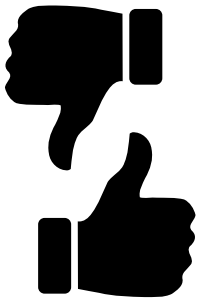
- Testing covered in other courses
- When to use asserts and when not?
- 💡 Idea: Asserts reflect invariants of a program
  - In a correct program the invariants would always be satisfied
  - A violation triggers an immediate program abort
- 👉 Asserts are good for testing
- 👉 Asserts are typically turned off for release build
  - No performance penalty

Improve Code Quality: Testing & Assertions I



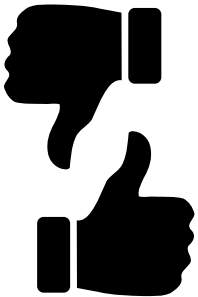
- Testing covered in other courses
- When to use asserts and when not?
- 💡 Idea: Asserts reflect invariants of a program
  - In a correct program the invariants would always be satisfied
  - A violation triggers an immediate program abort
- 👉 Asserts are good for testing
- 👉 Asserts are typically turned off for release build
  - No performance penalty
  - `-DNDEBUG` compiler flag

Improve Code Quality: Testing & Assertions I



- Testing covered in other courses
- When to use asserts and when not?
- 💡 Idea: Asserts reflect invariants of a program
  - In a correct program the invariants would always be satisfied
  - A violation triggers an immediate program abort
- 👉 Asserts are good for testing
- 👉 Asserts are typically turned off for release build
  - No performance penalty
  - `-DNDEBUG` compiler flag
  - ✋ Do not rely on asserts for security!

Improve Code Quality: Testing & Assertions II

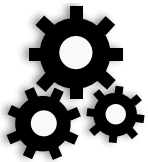


```c
#define TYPE_A   0
#define TYPE_B   1
#define TYPE_MAX 2

typedef struct {
    unsigned int type;
    size_t pos;
    int data[100];
} struct_t;

void internal_print(struct_t* s) {
  assert(s->type < TYPE_MAX); // type must always be valid
  size_t s_len =  sizeof(s->data) / sizeof(s->data[0]);
  assert(s->pos < s_len);   // pos must always be smaller than len
  for (size_t i = 0; i < s->pos; i++) {
      printf("Data: %d\n", s->data[i]);
  }
}
```

Improve Code Quality: Compiler Support I



- Compile with `-Wall -Wextra -Wconversion -pedantic` and fix all warnings

Improve Code Quality: Compiler Support I

- Compile with `-Wall -Wextra -Wconversion -pedantic` and fix all warnings
- Turn warnings into errors `-Werror`

Improve Code Quality: Compiler Support I

- Compile with `-Wall -Wextra -Wconversion -pedantic` and fix all warnings
- Turn warnings into errors `-Werror`
- Use compiler builtins properly, e.g. for integer overflow detection
  `https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html`

Improve Code Quality: Compiler Support I

- Compile with `-Wall -Wextra -Wconversion -pedantic` and fix all warnings
- Turn warnings into errors `-Werror`
- Use compiler builtins properly, e.g. for integer overflow detection
  `https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html`
- Use compiler's runtime sanitizers (`-fsanitize`)

Improve Code Quality: Compiler Support I

- Compile with `-Wall -Wextra -Wconversion -pedantic` and fix all warnings
- Turn warnings into errors `-Werror`
- Use compiler builtins properly, e.g. for integer overflow detection
  `https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html`
- Use compiler's runtime sanitizers (`-fsanitize`)
- Use static code analyzers `cppcheck` and `scanbuild`

Improve Code Quality: Compiler Support I

- Compile with `-Wall -Wextra -Wconversion -pedantic` and fix all warnings
- Turn warnings into errors `-Werror`
- Use compiler builtins properly, e.g. for integer overflow detection
  `https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html`
- Use compiler's runtime sanitizers (`-fsanitize`)
- Use static code analyzers `cppcheck` and `scanbuild`
- Fuzz your software (libAFL)

Improve Code Quality: Compiler Support II

Fortify

- `-D_FORTIFY_SOURCE=2`

Improve Code Quality: Compiler Support II

Fortify

- `-D_FORTIFY_SOURCE=2`
- Compiler adds extra checks to detect buffer overflows

Improve Code Quality: Compiler Support II

Fortify

- `-D_FORTIFY_SOURCE=2`
- Compiler adds extra checks to detect buffer overflows
- Compiler internally replaces calls to regular string functions with known-length string functions

Improve Code Quality: Compiler Support II

Fortify

- `-D_FORTIFY_SOURCE=2`
- Compiler adds extra checks to detect buffer overflows
- Compiler internally replaces calls to regular string functions with known-length string functions
- Not always possible

Improve Code Quality: More Resources

- CERT C Programming Language Secure Coding Standard

Improve Code Quality: More Resources

- CERT C Programming Language Secure Coding Standard
  - Exhaustive list of typical C-vulnerabilities

Improve Code Quality: More Resources

- CERT C Programming Language Secure Coding Standard
  - Exhaustive list of typical C-vulnerabilities
  - Memory, arrays, strings, integers, floating point, preprocessor, environment, I/O...

Improve Code Quality: More Resources

- CERT C Programming Language Secure Coding Standard
  - Exhaustive list of typical C-vulnerabilities
  - Memory, arrays, strings, integers, floating point, preprocessor, environment, I/O...
  - Rules, recommendations and example code

Improve Code Quality: More Resources

- CERT C Programming Language Secure Coding Standard
  - Exhaustive list of typical C-vulnerabilities
  - Memory, arrays, strings, integers, floating point, preprocessor, environment, I/O...
  - Rules, recommendations and example code
  - `https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard`

Improve Code Quality: More Resources

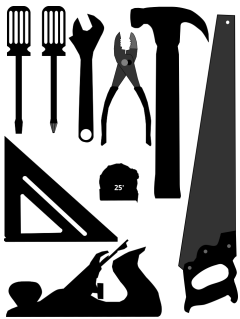
- CERT C Programming Language Secure Coding Standard
  - Exhaustive list of typical C-vulnerabilities
  - Memory, arrays, strings, integers, floating point, preprocessor, environment, I/O...
  - Rules, recommendations and example code
  - `https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard`
- OWASP Top 10 Web Application Security Risks

Improve Code Quality: More Resources

- CERT C Programming Language Secure Coding Standard
  - Exhaustive list of typical C-vulnerabilities
  - Memory, arrays, strings, integers, floating point, preprocessor, environment, I/O...
  - Rules, recommendations and example code
  - `https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard`
- OWASP Top 10 Web Application Security Risks
  - `https://owasp.org/www-project-top-ten/`

Summary & Outlook

Defensive Programming: Goals & Principles



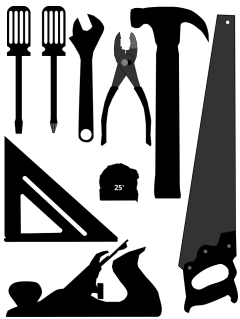
Sub-goals

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions
  - Compiler assistance

Defensive Programming: Goals & Principles



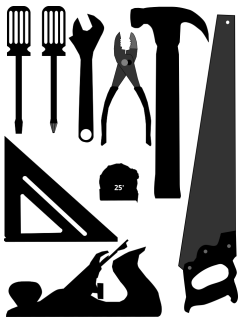
Sub-goals

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions
  - Compiler assistance

Defensive Programming: Goals & Principles



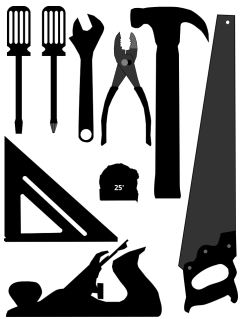
Sub-goals

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions
  - Compiler assistance

Defensive Programming: Goals & Principles



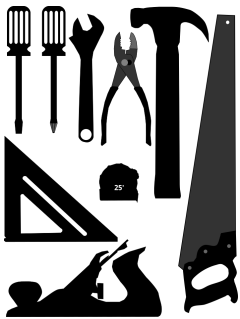
Sub-goals

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions
  - Compiler assistance

Defensive Programming: Goals & Principles



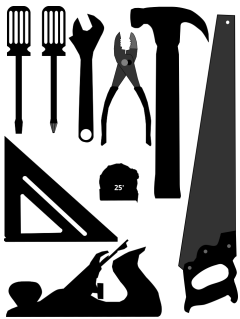
Sub-goals

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions
  - Compiler assistance

Defensive Programming: Goals & Principles



Sub-goals

- Memory safety
- Type safety
- Integer safety
- Secure data flow
  - Input sanitization
- Secure control flow
  - Error handling

General principles

- Choose appropriate language
- Improve code quality:
  - Coding standard
  - Source code reuse
  - Assumptions
  - Documentation
  - Testing & Assertions
  - Compiler assistance

Defense-In-Depth: Outlook

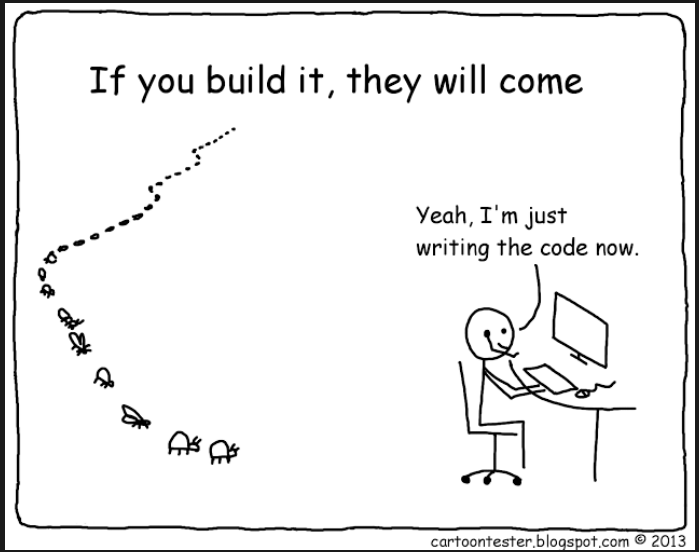
Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

- Vulnerability prevention (today) ☑
- Exploit prevention (next time)
- Privilege minimization (next time)

Questions?


If you build it, they will come
Yeah, I'm just writing the code now.
cartoontester.blogspot.com © 2013