

Secure Software Development

Defensive Programming

Daniel Gruss, Vedad Hadzic, Lukas Maar, Martin Schwarzl, Marcel Nageler

30.11.2022

Winter 2022/23, www.iaik.tugraz.at




1. Defense-in-Depth
2. Defensive Programming Overview
 - Safety Concepts
 - Secure Data Flow
 - Secure Control Flow
 - General Principles
 - Improve Code Quality
3. Summary & Outlook

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

Attacker's perspective

-  Vulnerability discovery
-  Exploitation
-  Privilege elevation (soon)

Defender's perspective

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

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

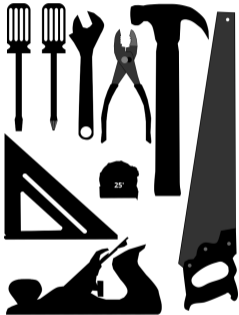


- ▮ Overall goal: eliminate **all** vulnerabilities
 - Proving a program's correctness is hard
 - Specification needs to be complete
 - Model-based Testing, Verification and Testing



- 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

- ❏ 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



- 🚩 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



🚩 Goal: eliminate temporal issues

👍 Use reference counting

- Avoids memory leaks and double free
- E.g., C++ smart pointers `std::shared_ptr` `std::weak_ptr`

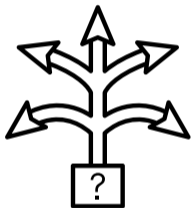
👍 Exclusive ownership: only one owner with write access

- 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;
```



🚩 Goal: eliminate type confusion issues

👍 Avoid unsafe casts

- Avoid cast to `(void*)` This should mainly be used for memory allocators
- Avoid C++ `reinterpret_cast`



- 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



- 🚩 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



🚩 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;
```




🚩 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 */  
}
```



- 👁 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**

JAVA SERIALIZATION BUG CROPS UP AT PAYPAL

by **Michael Mimoso**

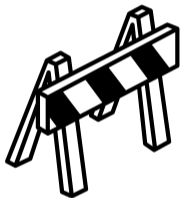
 Follow @mike_mimoso

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.

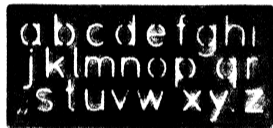


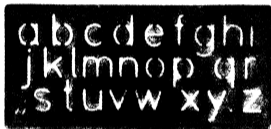
- ▮ Goal: sanitize dangerous input
 - Detect and reject
 - Filtering
 - Character escaping
 - **Use existing sanitizers wherever possible!**

👉 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  
...
```





💡 Idea: make parsing more uniform

👍 **Canonicalization before Sanitization!**

- Example: canonicalizing paths using `realpath`

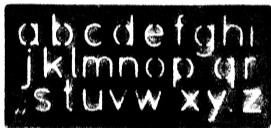
- Resolve relative paths

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

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

- Resolve symlinks:

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



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)



👉 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!**



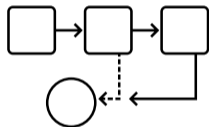
- Examples:
 - Shell: replace ``` with `\``
 - HTML: replace `<script>` with `<script>`

👉 Do not reinvent escaping!

👍 Use existing libraries

IMPACT

Secure Control Flow



- Return error codes
- Exception handling
- Goto



```
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 (line 3 & 4)
 - If there's nothing you can do in case of an error
 - In particular cleanup routines `fclose`, `munmap`, `(free)`



```
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



```
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

```
#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));
}
```





- 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/>

👉 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



- 👍 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 ;)



- 💡 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



```
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;
```

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



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



- 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 ...



💡 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

- 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

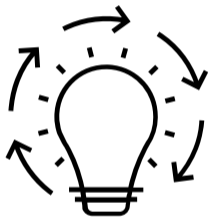


```
int readPassword(const char* path_password) {
    if (NULL == path_password) {
        return ERROR;
    }
    FILE* file_password = fopen(path_password, "r");
    if (NULL != file_password) {
        return ERROR;
    }
    ...
}
```



- 👍 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)
```



👉 “Do not rewrite your own \$THING”, especially

- Parsers
- Sanitizers
- Crypto libraries

👍 Reuse **established** libraries

- Stackoverflow/Copilot should be taken with care



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

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



💡 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...



- 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");
```



- 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



- 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!



```
#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]);
    }
}
```



- 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)

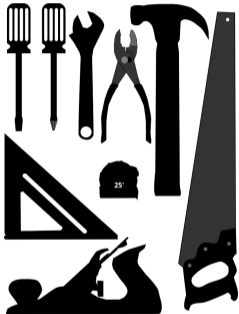


Fortify

- `-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

- 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






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

Attacker's perspective

-  Vulnerability discovery
-  Exploitation
-  Privilege elevation (soon)

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.

