SoK: Sanitizing for Security

Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, Michael Franz

University of California, Irvine
Finding Bugs in C/C++

Manual Analysis
- Code Review/Auditing

Static Analysis
- Clang Static Analyzer

Dynamic Analysis
- Valgrind (Dr. Memory)
  - AddressSanitizer
  - MemorySanitizer

C/C++ Source Code

Program Inputs
- Hand-written test suite
- American fuzzy lop
- libFuzzer
Finding Bugs in C/C++

Manual Analysis

Code Review/Auditing

Static Analysis

Clang Static Analyzer

Dynamic Analysis

Valgrind Dr. Memory

AddressSanitizer
MemorySanitizer

C/C++ Source Code

Program Inputs

Hand-written test suite

american fuzzy lop
libFuzzer
Dynamic Analysis Tools for C/C++

• More than 35 years of research in Dynamic Analysis Tools – often-called “Sanitizers” – that find vulnerabilities specific to C/C++
Exploit Mitigation vs. Sanitization (1/2)

**Attack Flow**

```
Integer Overflow → Heap Overflow + Function Pointer Overwrite → Indirect Call
```
Exploit Mitigation vs. Sanitization (1/2)

Exploit Mitigation Security Policies

Sanitization Policies

Memory Safety  Code Pointer Integrity  Control-Flow Integrity

Attack Flow

Integer \textbf{Overflow} \rightarrow \textbf{Heap \textbf{Overflow}} + \textbf{Function Pointer Overwrite} \rightarrow \textbf{Indirect Call}

UndefinedBehaviorSanitizer  AddressSanitizer  … and many others
Exploit Mitigation vs. Sanitization (1/2)

Exploit Mitigation Security Policies

Sanitization Policies

Memory Safety  Code Pointer Integrity  Control-Flow Integrity

Attack Flow

Integer Overflow → Heap Overflow + Function Pointer Overwrite → Indirect Call

UndefineBehaviorSanitizer  AddressSanitizer  … and many others
Exploit Mitigation vs. Sanitization (2/2)

<table>
<thead>
<tr>
<th></th>
<th>Exploit Mitigation</th>
<th>Sanitization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>The goal is to ...</strong></td>
<td>Mitigate attacks</td>
<td>Find vulnerabilities</td>
</tr>
<tr>
<td><strong>Used in ...</strong></td>
<td>Production</td>
<td>Pre-release</td>
</tr>
<tr>
<td><strong>Performance budget is ...</strong></td>
<td>Very limited</td>
<td>Much higher</td>
</tr>
<tr>
<td><strong>Policy violation leads to ...</strong></td>
<td>Program termination</td>
<td>Problem diagnosis</td>
</tr>
<tr>
<td><strong>Violations triggered at location of bug</strong></td>
<td>Sometimes</td>
<td>Always</td>
</tr>
<tr>
<td><strong>Tolerance for FPs is ...</strong></td>
<td>Zero</td>
<td>Somewhat higher</td>
</tr>
<tr>
<td><strong>Surviving benign errors is ...</strong></td>
<td>Desired</td>
<td>Not desired</td>
</tr>
</tbody>
</table>
Undefined Behavior in C/C++

- Buffer overflow
- Use-after-free
- Type errors
- Format string bug
- Signed integer overflow
- Null pointer dereferences
- etc.

**J.2 Undefined behavior**
The behavior is undefined in the following circumstances:

- Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).
- An object is referred to outside of its lifetime (6.2.4).
- A pointer is used to call a function whose type is not compatible with the referenced type.
- An object has its stored value accessed other than by an lvalue of an allowable type (6.5).
Undefined Behavior in C/C++

- Buffer overflow
- Use-after-free
- Type errors
- Format string bug
- Signed integer overflow
- Null pointer dereferences
- etc.

→ Well-known Security Vulnerabilities

**J.2 Undefined behavior**
The behavior is undefined in the following circumstances: ...

- Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).
  ...
- An object is referred to outside of its lifetime (6.2.4).
  ...
- A pointer is used to call a function whose type is not compatible with the referenced type.
  ...
- An object has its stored value accessed other than by an lvalue of an allowable type (6.5).
  ...
1. Memory and type safety violations vulnerable to memory exploits
2. Compilation of a program having UBs may result in vulnerable code

```
if (!tun)
  return POLLERR;
// privileged code
```

```
sk = tun->sk;
```

Null-pointer Dereference
2. Compilation of a program having UBs may result in vulnerable code

```c
if (!tun)
    return POLLERR;
```

Null pointer check gets eliminated (akin to CVE-2009-1897)
Security Implications of Undefined Behavior in C/C++ (2/2)

2. Compilation of a program having UBs may result in vulnerable code

```
if (!tun)
    return POLLERR;
```

```c
sk = tun->sk;
```

Null-pointer Dereference

```
mov rsi, QWORDPTR[rdi+8]
```

Privilege Escalation

Null pointer check gets eliminated (akin to CVE-2009-1897)
• **Most of these vulnerabilities** can manifest as memory and type safety violations.
• **Some UBs** may lead to unsafe code generation today.
• And, *things can change* as compiler optimizations evolve – called **time-bombs***

---

• **Some UBs** may lead to unsafe code generation today.
• **And, things can change** as compiler optimizations evolve – called *time-bombs*.

Sanitizer Design and Implementation

Bug Finding Technique

Program Instrumentation

(a) Language-level Instrumentation
(b) IR-level Instrumentation
(c) Binary Instrumentation

Metadata Management

(a) Dynamic Metadata
(b) Static Metadata

Object
Pointer
Instruction
Others
Sanitizer Design and Implementation: Bug Finding Techniques

Bug Finding Techniques

Class of Bugs
- Spatial Memory Safety Violation
  - Red-zone Insertion (Guard Pages)
  - Per-pointer Bounds Tracking
- Temporal Memory Safety Violation
  - Reuse Delay
  - Lock-and-key
- Use of Uninitialized Variables
  - Uninitialized Memory Read Detection
  - Uninitialized Value Use Detection
- Pointer Type Errors
  - Pointer Casting Monitor
  - Pointer Use Monitor
- Variadic Function Misuse
  - Dangerous Format String Detection
  - Argument Mismatch Detection
- Other Vulnerabilities
  - Stateless Monitoring

Bug Finding Technique
- Per-object Bounds Tracking
- Dangling Pointer Tagging
Sanitizer Design and Implementation: Program Instrumentation

Program Instrumentation

**Inlined Reference Monitor:**
Fine-grained run-time monitoring of program behavior to detect bugs as they occur.

- **(a) Language-level**
  - Source code (C/C++)
    - `check();`
    - `*ptr = 3;`

- **(b) IR-level**
  - Intermediate Representation (e.g., LLVM IR)
    - `call check`
    - `store ptr`

- **(c) Binary-level**
  - Binary
    - `11101110110101010110101011010101010101101010101011001101`

- **(d) Library Interposition**
  - `LD_PRELOAD` for instrumenting only calls to dynamically-linked external library functions
Sanitizer Design and Implementation: Metadata Management

Metadata Management

(a) Dynamic Metadata

Object
- Embedded before, after, and within an object

Pointer
- Fat pointer
- Tagged pointer

(b) Static Metadata

Instruction
- Result type of a casting operation
- Function type used in an indirect/variadic call

Others
- Class hierarchy
- Type aliasing information

Embedded
- Direct-mapped
- Multi-level
- Hash table

Disjoint
- Two-level trie
- Custom

Needs to be created and propagated at run time
Sanitizer Design and Implementation: Precision and Overheads

Program Instrumentation

(a) Language-level Instrumentation
(b) IR-level Instrumentation
(c) Binary Instrumentation

Metadata Management

(a) Dynamic Metadata
(b) Static Metadata

Object
Pointer

Instruction
Others

Bug Finding Technique

Spatial Memory Safety Violations
Temporal Memory Safety Violations
Use of Uninitialized Variables
Pointer Type Errors
Variable Function Monitor
Other Vulnerabilities

Red-zone Function Guard Pages
Per-pointer Bounds Tracking
Per-object Bounds Tracking
Data Race Monitoring
Debugging Failure Tagging

Use of
Uninitialized
Memory Read
Detection
Uninitialized
Value Use
Detection
Pointer Access Monitor
Pointer Use Monitor
Alignment Monitor

False positives
True negatives
False negatives
True positives

Bug Detection Precision and Compatibility

Performance and Memory Overheads

(b) Static Metadata

Instruction

Others
Our Analysis of Sanitizers

• Our analysis of 37 tools
  • We benchmarked 10 publicly available sanitizers on the same experimental platform (https://github.com/securesystemslab/sanitizing-for-security-benchmarks)

• Main observations
  • Performance is not a primary concern
  • Many false positives (marked as ) in tools other than widely-used ones such as ASan
  • Most  have partial coverage of bugs  
  • Widely deployed tools such as ASan have even smaller coverage
Well-defined programs \textit{w.r.t.} the ISO Standard

Programs conforming to the ISO standard
Precision: False Positives and False Negatives

Well-defined programs w.r.t. the ISO Standard

Programs conforming to the ISO standard
Well-defined programs w.r.t. the ISO Standard

Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object.*

But in practice it seems to be common to transiently construct out-of-bounds pointers.*

Precision: False Positives and False Negatives

Programs disallowed by the policy
- A typical sanitizer policy
- Programs conforming to the de facto standard
- Programs conforming to the ISO standard
Reducing Precision Gaps (1/2): Standard Compatibility

Compatibility with the ISO and de facto standards

Programs disallowed by the policy
A typical sanitizer policy
Programs conforming to the de facto standard
Programs conforming to the ISO standard
Reducing Precision Gaps (1/2): Standard Compatibility

• Many programs transiently construct OOB pointers (de facto standards)
  • Thus, supporting this code idiom increases a tool’s applicability

• Many tools, however, do not permit transient construction of OOB pointers
  • Some bounds checking tools invalidate pointers as soon as they go OOB
  • Dangling pointer tagging tools may incorrectly invalidate pointers, if OOB pointers are transiently stored in memory

• Many tools only support ISO standard compatibility by adding one byte between objects – this is not enough in practice
Reducing Precision Gaps (2/2): Finding Elusive Bugs

Finding bugs that elude existing or widely-deployed sanitizers

- Programs disallowed by the policy
- A typical sanitizer policy
- Programs conforming to the de facto standard
- Programs conforming to the ISO standard
Subclasses of memory safety violations that elude AddressSanitizer:

- Intra-object buffer overflow
- Buffer overflow into a valid but unintended object
- Uses of freed memory that are being reused

Type errors beyond bad casting (static_casts):

- C programs or C++ programs using C-style casts and reinterpret_casts
- Type errors are UBs that may silently break programs if does not instruct the compiler using flags like -fno-strict-aliasing
Reducing Precision Gaps (2/2): Finding Elusive Bugs

• Finding these elusive bugs, in general, requires \textbf{more precise dynamic metadata tracking}
  • Tracking \textbf{per-pointer metadata} such as pointer bounds
  • Tracking \textbf{effective types} of object storage in memory

• However, such metadata tracking poses precision and performance challenges
  • C’s weak type safety (e.g., pointer to integer casts, uses of void pointers) makes pointer metadata tracking difficult
  • The C standard has complex effective type and aliasing rules

• More research is needed in developing sanitizers that can find these elusive bugs while maintaining good compatibility
Fat pointer is not compatible with uninstrumented code

Disjoint pointer metadata can get outdated, when uninstrumented code updates a pointer without updating corresponding metadata.
Pointer Metadata Tracking Challenges: Pointer to Integer Casts

• Even with full instrumentation (via, e.g., dynamic binary translation), sanitizers can break programs having **pointer to integer casts**

  ![some_object_type * → uint64_t](image)

• Incompatible with fat/tagged pointers

• Disjoint pointer metadata can be a choice, but full pointer flow tracking across casts between pointers and integers can be expensive

• Existing tools stop tracking pointer metadata once they are cast to integers
• *Race-free* programs that use atomic operations can be problematic
  • Concurrent atomic operations from different threads without putting corresponding metadata operations into the same atomic unit can make metadata go out-of-sync

• Example of *naïve instrumentation*:

  Thread A

  1. `atomic_store(addr_of_ptr, ptrA);`


  Instrumented code

  Thread B

  2. `atomic_store(addr_of_ptr, ptrB);`


  metadata for `ptr` out-of-sync!
Type Error Detection Challenges

• Rules about determining an effective type of a stored value (i.e., effective type rules) are complex, due to weakly-typed nature of C
  • Prevalent uses of void pointer type and (omnipotent) character pointer type
  • `malloc` returns `void *`
  • `memcpy`-family of functions take and return `void *`
  • Type punning through C-style casts and union

• There are some tools that implement over-approximations of effective type rules, but precision and performance trade-offs are yet to be explored.

• Also, type error checking itself can be costly, because C’s aliasing rules permit a stored value to be accessed by using pointers of many different types
Other Future Research Directions

Composing sanitizers
• Can find bugs closer to their source without generating duplicated bug reports for the bug’s side-effects

Using hardware features to improve performance and compatibility
• e.g., Pointer tagging/memory tagging support in HW

Kernel and bare metal support
• Sanitizers for OS kernels, or non-user-space programs in general
Q & A

Thank you!

Dokyung Song
Ph.D. Student at UC Irvine
dokyungs@uci.edu