CS412 Software Security

Security Policies

Mathias Payer

EPFL, Spring 2019

Security Policies

A policy is a deliberate system of principles to guide decisions and achieve rational outcomes. A policy is a statement of intent, and is implemented as a procedure or protocol. (Wikipedia)

Isolation

Isolation is the process or fact of isolating or being isolated.

- Two components are isolated if their interactions are restricted.
- The OS isolates process from each other and only allows interaction through a well-defined API
    - Enabled through virtual memory and privilege domains

Least privilege

Least privilege requires that each component has the minimum amount of privileges to function.

- If any privilege is removed, the component stops to function

Compartmentalization

General technique of separating two or more parts of a system to prevent malfunctions from spreading between or among them.

- Requires combination of isolation and least privilege.
- Strong policy to contain faults to single components
- Forward reference: confused deputies are still a problem

Memory Safety

Memory safety is a property that ensures that all memory accesses adhere to the semantics defined by the source programming language.

Memory unsafety?

> *In short: I manipulate where the moving objects (sprites) are located or where they despawn, then I swap the item in Yoshi's mouth with a flying ?-block (thus the yellow glitched shell) and using a glitch (stunning) to spawn a sprite which isn't used by SMW and since it tries to jump to the sprite routine location, it indexes everything wrong and jumps to a place I manipulated earlier with the sprites (OAM) and because of the P-Switch it jumps to controller registers and from there the arbitrary code execution is started.*
> *Even shorter: Magic (by Masterjun3)*

`https://www.youtube.com/watch?v=OPcV9uIY5i4`

Practical memory safety

The gap between the operational semantics of the programming language and the underlying instructions provided by the ISA allow an attacker to step out of the restrictions imposed by the programming language and access memory out of context.

Memory unsafe languages like C/C++ do not enforce memory safety and data accesses can occur through stale/illegal pointers.

Memory safety

- Memory safety is a general property that can apply to a program, a runtime environment, or a programming language
  - See Mike Hicks: Memory Safety
- A program is memory safe, if all possible *executions* of that program are memory safe.
- A runtime environment is memory safe, if all runnable programs are memory safe.
- A programming language is memory safe, if all expressible programs are memory safe.
- Memory safety prohibits buffer overflows, NULL pointer dereferences, use after free, use of uninitialized memory, or double frees.

Memory unsafety (C/C++ view)

Memory safety violations rely on two conditions:

- Pointer goes out of bounds or becomes dangling
- The pointer is dereferenced (used for read or write)

Spatial memory safety

Spatial memory safety is a property that ensures that all memory dereferences are within bounds of their pointer's valid objects. An object's bounds are defined when the object is allocated. Any computed pointer to that object inherits the bounds of the object. Any pointer arithmetic can only result in a pointer inside the same object. Pointers that point outside of their associated object may not be dereferenced. Dereferencing such illegal pointers results in a spatial memory safety error and undefined behavior.

Spatial memory safety

```c
char *ptr = malloc(24);
for (int i = 0; i < 26 ++i) {
  ptr[i] = i+0x41;
}
```

- Classic buffer overflow
- Array is sequentially accessed past its allocated length

Temporal memory safety

Temporal memory safety is a property that ensures that all memory dereferences are valid at the time of the dereference, i.e., the pointed-to object is the same as when the pointer was created. When an object is freed, the underlying memory is no longer associated to the object and the pointer is no longer valid. Dereferencing such an invalid pointer results in a temporal memory safety error and undefined behavior.

Temporal memory safety

```c
char *ptr = malloc(26);
free(ptr);
for (int i = 0; i < 26; ++i) {
  ptr[i] = i+0x41;
}
```

Towards a definition

- Memory safety is violated if undefined memory is accessed, either out of bounds or deallocated.
- Pointers become capabilities, they allow access to a well-defined region of allocated memory. A pointer becomes a tuple of address, lower bound, upper bound, and validity.
  - Pointer arithmetic updates the tuple.
  - Memory allocation updates validity.
  - Dereference checks capability.
- Capabilities are implicitly added and enforced by the compiler.

Towards a definition

- Capability-based memory safety enforces type safety for two types: pointer-types and scalars
- Pointers (and their capabilities) are only created in a safe way.
- Pointers can only be dereferenced if they point to their assigned, still valid region

Memory safety: Java

- Replaces pointers with references (no direct memory access)
- No way to `free` data, implicit memory reuse after garbage collection
- Language and runtime system enforce safety, overhead?

Memory safety: Rust

- Strict type system and ownership implement memory safety
    - References are bound to variables
    - Clear ownership protects against data races: single mutable reference or zero or more immutable references
    - Variables that go out of scope are reclaimed
- Zero-cost abstraction

`http://theburningmonk.com/2015/05/rust-memory-safety-without-gc/`

Memory safety: C/C++

- How can we enforce memory safety for C/C++?
- What makes C/C++ memory unsafe?

Memory safety: C/C++

Two approaches:

- Remove unsafe features (dialect)
- Protect the use of unsafe features (instrumentation)

Memory safety: C/C++ dialects

- Extend C/C++ with safe pointers, enforce strict safety rules
- "Simple" approach: restrict C to safe subset (e.g., Cyclone)
    - Limit pointer arithmetic, add `NULL` checks
    - Use garbage collection for heap and region lifetimes for stack
    - Tagged unions (restricting conversions)
    - Normal, never `NULL`, and fat pointers (a fat pointer consists of address, base, size)
    - Exceptions and polymorphism to replace `setjmp`
    - Focuses on both spatial and temporal memory safety

Memory safety: C/C++ instrumentation

- Must track either pointers or allocated memory
- Check pointer validity when dereferencing
- Object based: cannot detect sub-object overflows, overhead for large lookups. Advantage that meta data is disjoint resulting in good compatibility
- Fat pointers: low compatibility due to inline metadata, can detect sub-object overflows
- Both object-based and fat-pointer based approaches fail to protect against arbitrary casts.

Memory safety: C/C++ SoftBound

Compiler-based instrumentation to enforce spatial memory safety for C/C++

- Idea: keep information about all pointers in *disjoint* metadata, indexed by pointer location
- Source code unchanged, compiler-based transformation
- Reasonable overhead of 67% for SPEC CPU2006

Memory safety: C/C++ SoftBound

```
char acctID[3];
char *id = &(acctID);
lookup(&id)->bse = &(acctID);  // store bounds
lookup(&id)->bnd = &(acctID)+3;
void acctInit() {
  char *p = id;  // local, remains in register
  char *p_bse = lookup(&id)->bse; // propagate
  char *p_bnd = lookup(&id)->bnd;
  do {
    char ch = readchar();
    check(p, p_bse, p_bnd);  // check
    *p = ch;
    p++;
  } while (ch);
}
```

Memory safety: C/C++ SoftBound instrumentation

1. Initialize (disjoint) metadata for pointer when it is assigned
2. Assignment covers both creation of pointers and propagation
3. Check bounds whenever pointer is dereferenced

Memory safety: C/C++ CETS

- Temporal memory safety is orthogonal to spatial memory safety
- The same memory area can be allocated to new object
- How do you ensure that a pointer references the new object and not the old object? How do you detect stale pointers?
  - Garbage collection?
  - Not reuse memory?

Memory safety: C/C++ CETS

CETS leverages memory object versioning. Each allocated memory object and pointer is assigned a unique version. Upon dereference, check if the pointer version is equal to the version of the memory object. Two failure conditions: area was deallocated and version is smaller (0) or area was reallocated to new object and the version is bigger.

Memory safety: C/C++ CETS

- Instrument memory allocation to assign a unique version to the memory area. Assign the same version to the initial pointer that is returned from the allocation function.
- Instrument memory deallocation to destroy the version of the associated memory area.
- Propagate version on pointer assignment
- Check if version between pointer and object match when dereferenced.

Type Safety

Well-typed programs cannot "go wrong". (Robin Milner)

Type-safe code accesses only the memory locations it is authorized to access.

- Different aspects of type safety: strong typed, weakly typed (implicit conversion)
- Static and dynamic type systems
- Lots of research, people still use C/C++ which are *not* type safe

C++ casting operations

`static_cast<ToClass>(Object)`

- Compile time check
- No runtime type information

`dynamic_cast<ToClass>(Object)`

- Runtime check
- Requires Runtime Type Information (RTTI)
- Not used in performance critical code

C++ static cast

```
Base *b = ...;
a = static_cast<Greeter*>(b);
```

C++ static cast

```
Base *b = ...;
a = static_cast<Greeter*>(b);

movq  -24(%rbp), %rax    # Load pointer
                         # Type "check"
movq  %rax, -40(%rbp)    # Store pointer
```

C++ dynamic cast

```
Base *b = ...;
a = dynamic_cast<Greeter*>(b);
```

C++ dynamic cast

```
Base *b = ...;
a = dynamic_cast<Greeter*>(b);
```

```
leaq  _ZTI7Greeter(%rip), %rdx
leaq  _ZTI4Base(%rip), %rsi
xorl  %ecx, %ecx
movq  %rbp, %rdi                # Load pointer
call  __dynamic_cast@PLT        # Type check
```

C++ virtual dispatch

```
class Base { ... };
class Exec: public Base {
  public:
    virtual void exec(char *prg) {
      system(prg);
    }
};
class Greeter: public Base {
  public:
    int loc;
    virtual void sayHi(char *str) {
      std::cout << str << std::endl;
    }
};
Greeter *greeter = new Greeter();
greeter->sayHi("Oh, hello there!");
```

C++ type confusion

```
class Base { ... };
class Greeter: public Base { ... };
class Exec: public Base { ... };

Greeter *g = new Greeter();
Base *b = static_cast<Base*>(g);
Exec *e = static_cast<Exec*>(b);  // type confusion
e->loc = 12;  // memory safety violation
e->sayHi();  // control-flow hijacking
```

C++ type confusion (full example)

```cpp
int main() {
  Base *b1 = new Greeter();
  Base *b2 = new Exec();
  Greeter *g;

  g = static_cast<Greeter*>(b1);
  g->sayHi("Greeter says hi!");
  // g[0][0](str);

  g = static_cast<Greeter*>(b2);
  g->sayHi("/usr/bin/xcalc");
  // g[0][0](str);

  delete b1;
  delete b2;
  return 0;
}
```

C++ type safety

- Keep type metadata for allocated objects (similar to memory safety)
- Check all casts dynamically
  - `static_cast<ToClass>(Object)`
  - `dynamic_cast<ToClass>(Object)`
  - `reinterpret_cast<ToClass>(Object)`
  - `(ToClass)(Object)`

HexType

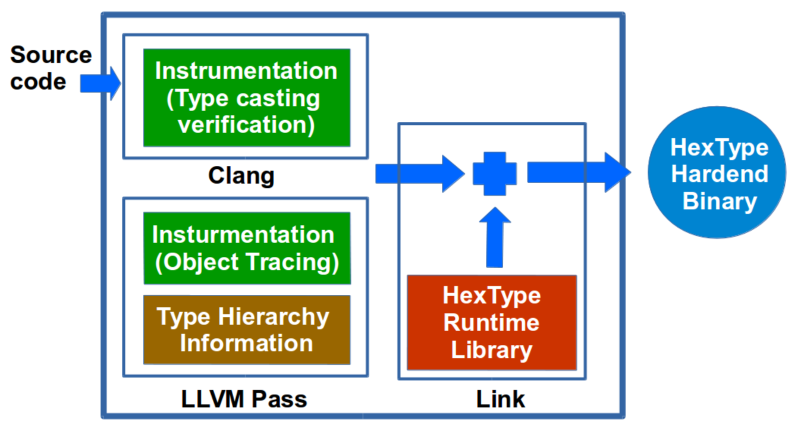
Source code
Instrumentation (Type casting verification)
Clang
Insturmentation (Object Tracing)
Type Hierarchy Information
LLVM Pass
HexType Runtime Library
Link
HexType Hardend Binary

Figure 1:

HexType

- Build global type hierarchy during compilation
- Instrument all forms of allocations, keep disjoint metadata
- Aggressively optimize
  - Limit tracing to “unsafe” types
  - Limit checking to unsafe casts
  - Replace dynamic cast with our check

Summary

- Memory and type safety are the root cause of security vulnerabilities
- Memory safety: distinguish between spatial and temporal memory safety violations
  - SoftBound: spatial memory safety through disjoint metadata for pointers
  - CETS: temporal memory safety through versioning
- Type-safe code accesses only the memory locations it is authorized to access
  - HexType: keep per-object type metadata and make all checks explicit
- SoftBound, CETS, and HexType are sanitizers: they trade performance for security during development (more on this later).
- Look at reading assignments