CMPSC 447: Future Directions

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Vulnerability

• Consists of these elements
  ‣ Flaw
  ‣ Accessible to an adversary
  ‣ Adversary has ability to exploit
Can We Really Reduce

• … Vulnerabilities and their exploitation?

• Directions of improvement
  ‣ Reduce/Eliminate Programming Flaws
  ‣ Reduce Accessibility
  ‣ Reduce/Eliminate Exploitability

• Take a look at the prospects of achieving such goals in the future today
Programming w/o Flaws

• Prevent flaws of all kinds
• Memory safety
  ‣ Spatial
  ‣ Type
  ‣ Temporal
• And others
  ‣ Filesystem
  ‣ Information Flow
Memory Safety

• Prevent safety violations from being possible
  ‣ In most cases, they are not possible
    • Most objects are only referenced by pointers in a safe way
  ‣ In others, we need some checking
    • Hopefully, via safe APIs
  ‣ But, is the checking correct?
Safety Validation

- **For memory safety in C**: CCured system proposed a method to identify the pointers only used in memory-safe ways (2002)
  - **Safe**: No pointer arithmetic (spatial) or type casting (type) operations
  - **Results**: Estimated 90% of pointers are only used in safe operations
  - **Problem**: Does not account for temporal errors
    - Under what conditions are temporal memory safety violations impossible by-design?
Type-Specific Pools

• **Hypothesis:** use type-specific allocation
  ‣ All objects and fields are aligned

• Type-specific pools
  ‣ Allocate an object of type A from a memory region containing only objects of type A
  ‣ Keep **data and pointers (fields) separate**
  ‣ Prevent **pointer-region mismatch**

• Must all references be of the same type? Default, yes
Step 1: Simple Safety Validation
CCured + Escape Analysis

72% of stack objects have no unsafe operations (are "safe")

Step 2: Collect Safety Constraints
For each memory error class

28% of stack objects have unsafe operations (may be "unsafe")

4% of stack objects do not have concrete safety constraints (assume "unsafe")

Step 3: Static Safety Validation
Value Range + Integer Range + Live Range

16% of stack objects validated statically (are "safe")

4% of stack objects do not have concrete safety constraints (assume "unsafe")

Step 4: Concolic Safety Validation
Def-Use Guided Concolic Execution

3% of stack objects validated concolically (are "safe")

5% of stack objects cannot be validated (assume "unsafe")

91.45% of stack objects protected by Safe Stack (without runtime checks!)

All Stack Objects

Possibility of Memory Unsafe DataGuard System (presented today at NDSS)
Memory Safety

• If a pointer may violate memory safety
  ‣ Need to enforce safety (at runtime)
  ‣ … Correctly
Enforcing Spatial Safety

- Two ways to enforce spatial safety
  - Check memory bounds
  - Automatic memory resizing

- Checking bounds
  - Make sure that a memory operation is limited to the associated memory region

- Automatic resizing
  - Resize the memory region to accommodate the memory required to satisfy the operation safely

- You now have APIs that check bounds and auto resize
Enforcing Bounds

• Enforce bounds checks
• int \textit{snprintf}(char *S, size_t N, const char *FORMAT, ...);
  ‣ Writes output to buffer S up to N chars (bounds check)
  ‣ Always writes ‘\0’ at end if N>=1 (terminate)
  ‣ Returns “length that would have been written” or negative if error (reports truncation or error)

• Thus, achieves goals of correct bounds checking
  ‣ Enforces bounds, ensures correct C string, and reports truncation or error
    • len = snprintf(buf, buflen, "%s", original_value);
    • if (len < 0 || len >= buflen) … // handle error/truncation

• What is needed for correctness?
Auto Resizing

• What about other functions like scanf?
  – scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf – all unsafe by default
  – Instead, use “%ms” to auto-resize
    – char *buffer = NULL; // Must be set to NULL
    – scanf(buffer, “%ms”);
  – Allocates memory for the buffer dynamically to hold input safely – null-terminated, no truncation required

• Note: also, can use for other functions that process input like getline
  – Should check whether the function you use supports this option
Safety from Type Errors

• **Type safety**
  ‣ Memory region is only referenced by pointers of one type
  ‣ Corresponding to the type of the memory region allocation

• **Memory safety** (for regions of multiple types)
  ‣ Memory region may be referenced by pointers of more than one type
  ‣ Semantics of all references correspond to allocation and consistent use of the memory region
  ‣ Think about “question” types in the project
Enforcing Type Safety

- Type casts create risks of type errors
  - Not type safe
- Any kinds of type casts guaranteed to be memory safe?
Enforcing Type Safety

• Type cast risk type errors
  ‣ Not type safe

• Any kinds of type casts guaranteed to be memory safe?
  ‣ Upcasts (spatial and type)
  ‣ Safe integer casts (same value, type) of same size (spatial)
  ‣ Other casts that preserve spatial and type constraints?

• Constraints – do not allow memory errors
  ‣ Ensure separation of data and pointers
  ‣ Ensure an access using a pointer will be within bounds
  ‣ May want more constraints (e.g., value)
Upcasts Are Memory Safe

- Only allow “upcasts” for type casts
  - An “upcast” from a child data type to a parent data type
    - Reduces fields – no overflow possible, fields are same type
  - Turn a downcast into an upcast – how?
    - If you can compute the set of types that may access a memory region
Tagged Casts Can Be Safe

- A **tagged union** is a data structure that has multiple, pre-defined types
  - Since we know the pre-defined sets of type for the memory region
  - We can limit the types of pointers that may access the memory region
  - And we can validate ahead-of-time that the combination of types is memory safe
    - E.g., pointer fields are only aligned with pointer fields
- **Problem**: Need to find set of pre-defined types
Safety from Temporal Errors

• Type-specific pools
  ‣ Like type safety
    • Memory region is only referenced by pointers of one type
    • Corresponding to the type of the memory region allocation
  ‣ Like “compatible” tagged unions
    • Could exploit type-specific pools for a compatible set of pre-defined types
    • Multiple types that comply with memory safety requirements

• Otherwise
  ‣ Zeroing pointers at initialization and deallocation seems easiest – can add up as overhead
Detecting Vulnerabilities

• (1) Using safe APIs
• (2) And having program analyses to detect flaws
  ‣ Fuzzing, static analysis, symbolic execution
• What would you need analyses for?
Programming Safely

• (1) Using safe APIs
• (2) And having program analyses to detect flaws
  ‣ Fuzzing, static analysis, symbolic execution
• What would you need analyses for?
  ‣ Even use of safe APIs and techniques may be incorrect
static int queue_manag(void *data)
{
    /* backlog is declared without initialization */
    struct crypto_async_request *backlog;
    if (cpg->eng_st == ENGINE_IDLE) {
        backlog = crypto_get_backlog(&cpg->queue);
    }
    /* Uninitialized backlog is used*/
    if (backlog) {
        /* uninitialized pointer dereferenced! */
        backlog->complete(backlog, -EINPROGRESS);
    }
    return 0;
}
Static Analysis for UBI

- Implementation:
  - LLVM 7.0.0
  - 13K+ LoC
  - SE Engine: KLEE
Limiting Access to Flaws

• If programs may still have flaws, how do we reduce the ability of an adversary to access them?
Limiting Access to Flaws

• If programs may still have flaws, how do we reduce the ability of an adversary to access them?
  ‣ Attack surface
    • Limit the places where adversary input is allowed
• **Insight:** Only a small fraction of system calls expect to use adversary-controlled input
• **Insight**: Only a small fraction of system calls expect to use adversary-controlled input

- Any new attack surface is often the source of vulnerabilities
**Attack Surface**

- **Insight:** Only a small fraction of program system calls expect to use adversary-controlled input

- Limit the system call to only access “safe” objects

- What is “safe”?
Attack Surface

• **Insight**: Only a small fraction of program system calls expect to use adversary-controlled input

• Limit the system call to only access “safe” objects

- Apache `httpd`
- `httpd_t`
- `mmap()`
- `open()`
- `read()`
- `accept()`

• What is “safe”? Not modifiable by an adversary
Limiting Exploitability of Flaws

• If programs may still have flaws that adversaries can access, how do we reduce the ability of an adversary to exploit them?
  ‣ Isolation
    • Isolate good data from bad
  ‣ Restriction
    • Limit targets to which a compromised pointer can reference
Isolation

• Isolate data that is safe from memory errors from other unsafe data
  ‣ Only safe memory references possible for all safe objects

• Unsafe memory references are possible via unsafe pointers
  ‣ But, if safe objects are not accessible from those unsafe memory references then they are protected
Motivation for Partitioning

- Split the application into multiple partitions
- Each partition is isolated using some isolation mechanism such as OS processes

Although some partition of a program has been hijacked, sensitive data can still be protected
Multi-Stack (Safe Stack)

- **A separate stack region** for objects validated to be safe from spatial errors (**Safe Stack**)

- **Results:** Safe stack objects are protected from spatial errors without runtime checks

- **With DataGuard** all objects on the safe stack have been proven safe from all three classes of memory errors

- **Can do same kind of thing with heap objects as well!**

- **But, isolation between stacks is currently implemented by ASLR**
• Limit memory accesses only to legal values
  ‣ Any example of this approach you can recall?
Restriction

• Limit memory accesses only to legal values
  ‣ Any example of this approach you can recall?
  ‣ CFI – restrict targets of an indirect call to the CFG
  ‣ SFI – restrict targets of a memory access to a region
  ‣ Privilege separation restricts accesses to the memory regions associated with a subset of functions (code) and their data

• How does SFI work?
SFI Policy

Fault Domain

Code Region (readable, executable)

Data Region (readable, writable)

1) All jumps remain in CR
2) Reference monitor not bypassed by jumps

All R/W remain in DR [DB, DL]
Take Away

• **Reducing vulnerabilities** is the target of defenses

• We can reduce flaws
  • But, need help in validating safe cases and/or identifying cases helpfully – e.g., analysis

• We can limit accessibility to flaws further
  ‣ Attack surfaces and privilege separation

• We can reduce the ability of adversaries to exploit the remaining flaws
  • May be a bit expensive w/o hardware help or need to be more targeted