Software Security III: Memory Errors - More Attacks & Defenses

Chengyu Song

Slides modified from Dawn Song
Memory error: attacks & defenses

- Best solution: type safe languages and secure coding practice
- Three widely deployed exploit prevention techniques
  - Stack canary, DEP, ASLR
- And how they can be bypassed
More memory errors

• Q1: any other bugs can also cause out-of-bound access?
• Q2: any other memory errors other than out-of-bound access?
Incorrect bound checks

/* signed vs. unsigned, int base[], int size, int index */
/* lower bound */
if (index < size) return base[index];
/* upper bound */
memcpy(base, src, index);

/* off-by-one, char str[] */
for (int i = 0; i <= strlen(str); i++)
    str[i] = 'a';
for (int i = strlen(str); i >= 0; i--)
    str[i-1] = 'a';
Type away (1)

• Pay additional attention to signed vs. unsigned integer
  • DO NOT ignore compiler warnings
  • Use `size_t` when not sure
• Always remember: index starts from 0, last element's index is `size-1`
**Integer overflow/underflow**

- Integers have limited range (check `limits.h`)
  - Will wrap around

  ```c
  int si1 = 0x7fffffff, si2 = si1 + 1;
  • si2 = ??, si2 - 1 = ??
  ```

  ```c
  unsigned int ui1 = 0xffffffff, ui2 = ui1 + 1;
  • ui2 = ??, ui2 - 1 = ??
  ```

- Affected operations: addition, subtraction, multiplication
/* Causing incorrect check */
int *base, *end; size_t length; */
if (base + length < end) memcpy(base, src, length);

/* IO2BO */
size_t x, y; */
char *buf = malloc(x*y);
Take away (2)

- Check for integer overflow/underflow is extremely tricky!!
  - a*b*c*d
  - if (a*b/b != a), signed vs. unsigned
- Utilize compiler builtins when possible
  - Example: __builtin_smul_overflow(a, b, &result)
  - http://clang.llvm.org/docs/LanguageExtensions.html#checked-arithmetic-builtins
Type confusion

```c
char a;
void *p = &c;
*(int *)p = 1;

struct base {int a;};
struct d1 : base {int b;};
struct d2 : base {int c; int d;};

base *b = new d1;
d2 *d = static_cast<d2*>(b);
d->d = 1;
```
Type confusion (cont.)

- Type of the pointer mismatches type of the memory object
  - Arbitrary casting
  - Down casting
  - use-after-free

- Solution? Runtime type information (RTTI)
  - No perfect implementation yet
  - `dynamic_cast` is more secure but is very slow
Temporal memory errors

- So far we've been focusing on **spatial memory errors**
- Another type of memory errors is **temporal errors**
  - **Use-after-free**
  - **Use-before-initialization**
Use-after-free (UAF)

/* allocation */
void *p = malloc(size);
/* valid use */
memset(p, 0, size);
/* free */
free(p); /* <--- p is called a dangling pointer now */
/* invalid use */
int i = *(int *)p;
How to exploit UAF?

- Through **overlapping** (instead of overwriting/overreading)
  - Allocation, free, re-allocation, use
- Why would overlapping happen?
  - Heap allocators tend to reuse recently freed memory to speed up allocation
Take away (3)

- Nullify pointers once freed
  - Cannot solve all UAF, but can prevent many
- Use reference counter when necessary
  - Ref counter can go wrong as well
  - Use languages support whenever available, e.g., smart pointers
- Consider conservative garbage collection
  - MemGC in MS Edge
Use-before-initialization (UBI)

```c
int i;
printf("%d", i);
```

- What would be the printed value?
How to exploit UBI

• Similar to UAF, through overlapping
  • UAF: type of the old pointer mismatches type of the new object
  • UBI: type of the new pointer mismatches type of the old object

• How to overlap?

• Why would the initialization be missing?
  • Developer's mistake
  • Compiler's "mistake"
Take away (4)

• Remember to explicitly initialize important variables/data structure fields
• Remember to clear (e.g., `memset(0)`) sensitive information in your buffer before free
Additional attack targets

- What we’ve discussed? return address, frame pointer
- What else?
  - Function pointers
  - vtable (virtual function table) pointers
  - Any pointer
  - Any conditional branch dependencies (a.k.a. data-only attack, data-oriented attack)
Information leak

• Besides executing arbitrary attacker-controlled logic, leaking out critical information is also a way to attack
  • Crypto keys: e.g., HeartBleed
  • Randomness: ASLR, stack canary, etc
  • Screen
Any news on the defense side?

- Control-flow Integrity (CFI)
- Hardware-assisted bound check (Intel MPX)
The CFI security policy dictates that software execution must follow a path of a Control-Flow Graph (CFG) determined ahead of time. -- Abadi, et al. [1]

What is a control-flow graph?

- Vertex: basic block (what is this?)
  - A straight-line code sequence with no branches in except to the entry and no branches out except at the exit.
- Directed edge: control transfer
- Forward edge: call, jump
- Backward edge: return
CFG (cont.)

• How to construct the CFG?
  • Statically: source code, binary
  • Dynamically
CFI in compilers

- Microsoft: control-flow guard (`/guard:cf`)
  - Windows 8.1 and VS 2015 and newer
  - Return flow guard
- GCC: vtable verification (VTV)
- Clang: `-fsanitize=cfi`
  - https://clang.llvm.org/docs/ControlFlowIntegrity.html
Effectiveness?

- Control-flow hijacking becomes much harder
  - Can still be bypassed due to imprecision of CFG
- But provides no protection against other types of exploit techniques
  - Data-only attacks
  - Information leak attacks
Summary

- Two type of memory errors: spatial and temporal
- Can be caused by many types of bugs
  - Hard to eliminate and are still popular
- Can be exploited in many ways
  - Hard to prevent
- Secure code practices and compiler options
For next class ...

- Software Security IV: fuzzing