CITS3007 Secure Coding
Memory and arithmetic errors

Unit coordinator: Arran Stewart

Outline

- Buffer overflows
  - relevance, related vulnerabilities, protections
- Integer overflows and overflows

Buffer overflows

Buffer overflows – relevance

- We've seen a historical case where buffer overflows were used in a security incident (the Morris Internet worm)
- Buffer overflows are *still* a very major source of vulnerabilities
- The CWE ("Common Weakness Enumeration") database has annual "Top 25 most dangerous software weaknesses" lists – it's based on an analysis of the CVE database, weighting particular vulnerabilities by their prevalence and severity
  - (The CWE is a classification of types of vulnerabilities – like a dictionary or glossary. Not to be confused with the CVE, "Common Vulnerabilities and Exposures", a database of publicly disclosed flaws in software programs.)
  - CWE-787, "Out-of-bounds write", the category to which buffer overflows belong, has been in the top 2 CWEs for 3 years running

Buffer overflows – relevance

They can persist for a very long time, undetected

- The "Baron Samedit" vulnerability is a heap-based buffer overflow vulnerability (CVE-2021-3156) in the `sudo` program, which is ubiquitous on Unix-like systems
  - the bug was introduced in 2011, and not detected until 2021
  - By calling the `sudoedit` command (a symlink to `sudo`) with specially crafted arguments, an attacker could execute arbitrary code and gain root privileges.
- The "BootHole" vulnerability in Linux's GRUB2 bootloader also arose from a buffer overflow
  - It was present from the very first version of GRUB2 (released in 2010) until patched versions were released in 2020
  - It allowed an attacker to control the "secure boot" process (which normally can't be done from within the booted OS at all, even by a superuser)

Related vulnerabilities

- CWE-787, "Out-of-bounds Write", includes as sub-types of vulnerability:
  - CWE-121 Stack-based Buffer Overflow: A buffer on the stack is overflowed (can overwrite stack return addresses)
  - CWE-121 Heap-based Buffer Overflow: A buffer on the heap is overflowed (can corrupt data)
  - CWE-823 Use of Out-of-range Pointer Offset: Pointer could potentially point anywhere in memory

Underlying causes

- a buffer (array or string) is just some space in which data can be stored.
- some languages check at runtime whether a reference to an array position is in bounds, others don't
  - C does not; Java and Python do
  - In C++, bounds checking may be provided – e.g. if using the `std::vector` class, an alternative to access syntax `myvec[42]` is to use `myvec.at(42)`
- If, while writing to a buffer, a program overruns the bounds of the buffer, then that's a buffer overflow.
  - (Normally, this is due to copying data past the "right-hand end" of the buffer, but there's no reason in principle why it might not go the other way.)
- If the data overwrites adjacent data or program instructions, that can lead to unpredictable behaviour and security vulns.

Types of buffer overflow

- stack buffer overflow – overrun a buffer declared as a variable on the stack. Attackers would rely on the fact that this would typically overwrite adjacent variables and/or program instructions
- heap overflow – we overrun a buffer contained in dynamically allocated memory. Attackers would rely on the fact that this would overwrite other data structures stored on the heap.

Mechanics of overflow

- The classic way to exploit memory-bounds vulnerabilities is to do *code injection*
  - Malicious code is put into some predictable location in memory – typically, somewhere where *data* would normally stored
  - Then the vulnerable program is tricked into executing that code (e.g. by overwriting the return address of the stack frame).
- But there are other ways to exploit vulnerabilities without code injection.
  - you could corrupt data – e.g. you might overwrite a variable that's used to select a branch of an `if` statement

Preventing buffer overflow vulnerabilities

- Re-write in a memory safe language (Java, Python)
- Audit/static analysis
- Prevent execution of injected code
- Add runtime instrumentation to detect problems
- Make it harder for attackers to exploit code and data through address randomisation
- Testing
- Validate untrusted input (discussed in future lectures)

Prevention – memory safe language

Re-write in a memory safe language (Java, Python)

- Not always possible for existing code
- Memory-safe languages may have their own disadvantages (slower, slow start-up time, need to distribute a large runtime, possibly harder to find qualified developers, lack portability of C)

Prevention – audit/static analysis

- Buffer overflows (and other attacks relying on “wild pointers”) tend to arise from format string vulnerabilities and common errors in managing dynamic memory
- So trying to eliminate those sources of errors goes a long way to eliminating the problem
- Manual audits and automated static analysis can be applied to find such errors

Prevention – runtime instrumentation

It may be possible to add run-time checks to a normally unchecked language.

- May be in the form of a library or alternative implementation of standard functions (e.g. `malloc`, `strncpy`, etc)
- Compilers such as `gcc` and `clang` offer *sanitizers* which can be enabled by providing flags at compilation time (e.g. `-fsanitize=undefined` is an umbrella "undefined behaviour sanitizer" for `gcc`)
  - These sanitizers can detect errors such as buffer overflows

Address sanitizer

An example sanitizer – AddressSanitizer, originally developed by Google.

A refinement of earlier techniques (e.g. "Electric Fence", developed by Bruce Perens in 1987 while working at Pixar).

It replaces the normal `malloc` and `free` functions with versions where the memory around `malloc`-ed regions is "poisoned".

Reads and writes are checked to make sure they're not using addresses in the "poisoned" regions – if they are, the program aborts.

Stack "canaries"

Another runtime checking approach is to embed "canaries" in stack frames

- Canaries can be e.g. random strings chosen at program startup
- Code is inserted that verifies the integrity of the canaries prior to function return
- If an attacker overflows a stack buffer, they won't know the correct value of the "canaries", so the overflow will be detected

Runtime instrumentation limitations

In general, any of these techniques will reduce performance (due to additional memory being required and/or additional runtime checks being performed)

- However, the cost may be tolerable
  - e.g. Use of StackGuard (stack canary technique) results in approx 8% performance penalty

Prevention – address randomization

This technique is called **ASLR** (Address Space Layout Randomization)

Used to prevent an attacker from reliably jumping to some particular function/address in memory.

- We start the stack and heap at some random location in memory
- We map shared libraries to random locations in process memory
  - This means that the attacker can no longer e.g. jump directly to the `system` or `exec` function

Prevention – validate untrusted input

We need to be particular careful when we're reading and using data (e.g. especially sizes or lengths of things) from potentially untrusted sources. e.g.

- over the network
- from a user-supplied file

Audit/static analysis

Programs can be manually or (partly) automatically checked for common problems:

- Use of “unsafe” functions
- Improper use of “safe” functions
- Poor memory management practices

Static analysis tools:

Examples include Splint, OCLint, Clang Static Analyzer

We will see more on these in labs.

Unsafe library functions

Many C string functions are *unsafe* to use because they rely solely on the `NUL` delimiter to mark the end of strings; so if this delimiter is missing, they will keep reading or writing memory til a `NUL` is encountered.

These functions include:

- `strcpy (char* dest, const char *src)`
- `strcat (char *de st, const char *src)`
- `gets (char *s)`
- `scanf (const char *format, ...)`
- and many more.

In general, the "safe" equivalents of those functions should be used.

"Safe" library functions

e.g. `char *strncpy(char *dest, const char *src, size_t n)` is a "safe" version of `strcpy (char* dest, const char *src)`.

However, **the word "safe" here is a misnomer**. They are definitely *safer* than the originals, but still need to be used properly.

`strncpy` will copy at most `n` characters; but it *won't* properly terminate `dest` unless a `NUL` appears in those `n` characters.

So the proper use is usually something like:

```
#define BUF_SIZE 50
char buf[BUF_SIZE];
strncpy(buf, src, BUF_SIZE);
buf[BUF_SIZE-1] = '\0';
```

“W xor X” (“write XOR execute”)

- Modern CPUs provide hardware support for marking segments of memory as non-executable
  - Both AMD and Intel processors support this
- The stack and heap can be put in non-executable memory
  - Any attempt to execute memory in those regions will result in a “fault”

"W xor X"

- Marking memory as non-executable does *not* prevent data structures or return addresses from being corrupted
- It's possible to overwrite a stack return address with some library routine, and arrange the contents of the frame above it to look like arguments to that routine
- So the attacker cannot execute arbitrary code within the running process; but they may still be able to call functions like `system` (which executes commands via the operating system's shell).

return-oriented programming

- Marking memory as non-executable doesn't defend against a style of attack called "return-oriented programming".
- A "return address" can point to any sequence of instructions ending in a "return" (called "gadgets")
- Therefore, it's possible to arrange the stack such that stack frames will execute a sequence of these gadgets, with appropriate data acting as function arguments
- If an attacker can find appropriate "gadgets" in library routines, they may be able to perform arbitrary computations without needing to inject code

Integer overflows and underflows

Integer overflows and underflows

Informally, "overflow" tends to be used to describe several different phenomena.

- Intended "wraparound" of integer types in various languages
- "Underflow" – wraparound from the bottom
- Exceeding the bounds of numbers representable in an integer type, resulting in undefined behaviour
- Assigning a number to a type to small too hold it, resulting in "truncation"

Any of these can result in security vulnerabilities, due to a number not holding the value programmers expect it to hold.

Causes

Most people are used to thinking of numbers as if they were idealized mathematical integers.

For two such integers x and y, if $x > 0$ and $y > 0$, then $xy > 0$, $xy > x$ and $xy > y$.

But for (say) an `unsigned char`, we have $13 \times 20 = 4$.

And for a `signed char`, the behaviour is undefined (but probably, $10 \times 13 = -126$).

Summary

- unsigned integer types: If a new value is out of bounds, wrap around
- signed integer types: If the new value is out of bounds, undefined behaviour. Unpredictable, but often the new value will wrap around
- conversion from a larger type to unsigned integer type: Wrap around (truncation)
- conversion from a larger type to signed integer type: Implementation defined, but typically will truncate

Unsigned integer wraparound

- In C, for *unsigned* integer types, their intended behaviour is that if you attempt to calculate a value that would go outside their bounds, the value will "wrap around"
  - i.e., if the maximum representable number is N, then trying to create the value $N + m$ will instead give the value $N \mod m$.
  - And likewise, values will wrap around if you try to create a value less than 0

Signed integer overflow

- For *signed* integer types, exceeding the representable bounds for a type results in undefined behaviour.
- In practice, on many platforms the value will "wrap around".
- **However**, the compiler is allowed to *assume* that the value *hasn't* wrapped around. (That's what "undefined behaviour means": only programs with no UB have a well-defined meaning; so the compiler is allowed to assume that no UB ever occurs.)

Signed integer overflow

- The consequence is that once UB has occurred, you *can't reliably check for it*.

  e.g. Suppose we have an `int n` containing some positive number, and we add one to it. How can we check to see if it overflowed?

  Maybe we save the value of old `n`, and make sure `n > old_n`; or ask whether `n < 0`.

  But the **compiler is allowed to tell us** that `n` **is** greater than `old_n`, because **that's what would be true** if no UB occurred.

  It can "optimize away" the results of checks like `n > old_n`, because it "knows" they must evaluate to `true`.

Conversion between types

If you assign a larger integer type to a smaller *unsigned* type, the result will just be modulo'd with the `MAX + 1` for that type until the result is in range.

The effect is to *truncate* the value.

For example:

```
unsigned int  a = 0x10003;
unsigned char b = a;
```

After the statements above are executed, `b` will be equal to 3.

Conversion between types

If you assign a larger integer type to a smaller *signed* type, the result is implementation-defined (and can include raising an implementation-defined signal which would terminate the program).

Typically, this too will result in truncation.

```
signed int  a = 0x10003;
signed char b = a;
```

After the statements above are executed, `b` will (probably) be equal to 3.

Vulnerabilities arising from truncation

```
struct thing_t {
  unsigned short len;
  char * buf;
};

void myfunc() {
  size_t len = get_size();
    // get len from e.g. argv,
    // or a network message
  struct thing_t thing;
  thing.buf = malloc(len + 3);
  thing.len = len;
  // Suppose len is USHRT_MAX+10.
  // Then thing.len is incorrectly
  // set to 13
```

```
// Later, the program might use thing:
const size_t BUF_SIZE = 100;
char buffer[BUF_SIZE];
if (thing.len < BUF_SIZE) {
  strcpy(buffer, thing.buf);
  // overflow, as thing.buf is actually
  // much bigger
}
```

OpenSSH integer overflow vulnerability

This results from unexpected wraparound in `size_t`.

- See https://nvd.nist.gov/vuln/detail/CVE-2002-0639

Vulnerable code is in the function `input_userauth_info_response`.

```
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}
```

OpenSSH integer overflow vulnerability

```
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}
```

- `nresp` can be attacked-controlled (it means "number of responses").
- So set `nresp` to `(SIZE_MAX + 1) / 4`, where `SIZE_MAX` is the largest value a `size_t` can hold – $2^{64} - 1$, on my machine – so for me `nresp` will be 2^{62}.
- Arithmetic on a `size_t` is done module 2^{64}.
- So `nresp * sizeof(char*)` will be `nresp * 4` (mod 2^{64}), which is 0.
- 0 is a valid argument to `malloc` (though it won't actually allocate any memory).
- So `response` will succeed, allocating no memory, and the subsequent loop will immediately overflow the `response` buffer, corrupting data on the heap.

Defending against integer overflow

- Use appropriate types
- Do arithmetic in a wider type
- Use compiler flags
- Use libraries or code that provide “safe” arithmetic functions

Use appropriate types

- Need a size or a (non-negative) count? Use `size_t`
- Need a specific bit-width? Use `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, etc.
- Need an integer which will hold any pointer? Use `intptr_t`

Use compiler flags

- `-fwrapv` – Treat signed integer overflow behaviour as well-defined – it "wraps round"
- `-ftrapv` – With `gcc` on AMD64, supposed to cause the `SIGABRT` signal to be raised, which will normally end the program. But apparently is broken (see https://gcc.gnu.org/bugzilla/show_bug.cgi?id=35412)
- `-fsanitize=signed-integer-overflow` – print error report and continue
- `-fno-sanitize-recover=signed-integer-overflow` – print an error report and exit the program;
- `-fsanitize-trap=signed-integer-overflow` – raise a trap (usually, the `SIGABRT` signal)

“Safe” arithmetic

```
if (a > 0 && b > INT_MAX − a)
  abort();
if (a < 0 && b < INT_MIN − a)
  abort();
result = a + b;
```

"Safe" arithmetic

To do signed wraparound, if there's no compiler support:

- Convert from signed `char` to unsigned. (You can just write: `unsigned char myuchar = my_signed_val`.) The value will wrap as necessary.
- Do your calculations on the unsigned numbers. The results will always be well-defined.
- Convert back to `signed char`, in this way: if the unsigned result (call it `res`) is less than or equal to `SCHAR_MAX`, we're fine.

  If it isn't: modulo the unsigned result with `SCHAR_MAX`, and add it to `SCHAR_MIN`.

Integer bounds in other languages

- Java *only* has signed integer types – no unsigned types. The behaviour of all types is that they "wrap" around if overflow would occur.

  Since Java 8, it provides methods like `Math.addExact()`, which will throw an exception if overflow or underflow would occur.

  - The JVM can still suffer from overflow errors in underlying C++ code – e.g. see https://bugs.openjdk.org/browse/JDK-8233144

- Treatment of integers in Python varies from version to version.

  Typical approach is: use the underlying C `int` type by default; however, if a value would exceed the bounds of an `int`, it gets automatically promoted to an arbitrary-precision integer type

  - How to detect overflow? One way: do the C arithmetic as normal, then try it with the C `int`s cast to `double`s. If the result differs greatly, then underflow or overflow occurred.

Overflow CWE

- Has a CWE ID – CWE-190 Integer Overflow or Wraparound
- a calculation can produce an integer overflow or wraparound, but the program logic assumes the new value will always be larger than the old
- if the integer is got from a user/attacker, they may be able to deliberately trigger this with user-supplied inputs
- if the integer is then used to control looping, make a security decision, allocate memory etc then the vulnerability becomes critical