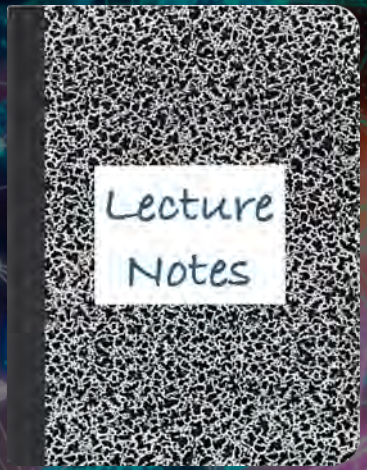
CS 419 – Computer Security

Week 7: Recitation
Review and Format-String Bugs



Paul Krzyzanowski

© 2026 Paul Krzyzanowski. No part of this content may be reproduced or reposted in whole or in part in any manner without the permission of the copyright owner.

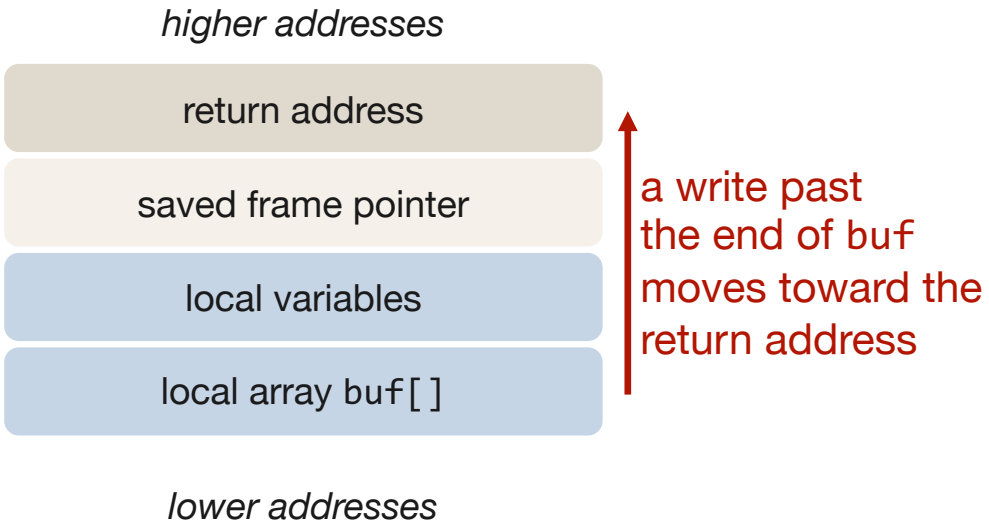
Part 1
Review of the Key Concepts

What a Memory-Safety Bug Is

A memory-safety bug lets a program read or write outside the bounds or the lifetime of an object.

Spatial errors	**Temporal errors**
Outside the bounds of an object	*Outside the lifetime of an object*
Buffer overflow writes past the end of an array	Use-after-free uses memory after it is freed
Out-of-bounds read reads past the end of an object	Double-free frees the same block twice

The Stack and Control Flow



Reaches the return address → **control-flow hijacking**: the return jumps wherever the attacker wrote

Changes a nearby decision variable, leaving the return address → **data-only attack**: normal code, wrong result

Why does writing past the end of a local array move toward the return address?

Reading Past a Buffer

An **out-of-bounds read** accesses memory beyond an object.
The danger depends on what the program does with the result.

The program sends the bytes back → it discloses whatever sat in adjacent memory

The *read* touches an unmapped page → the program crashes

A leaked pointer reveals a real address inside the process.
This will expose the address of a randomized region of memory and undermine ASLR (address space layout randomization) — a defense we review later

The Heap and Lifetime Errors

Heap writing attacks:
Common targets are the allocator's bookkeeping and the pointers stored inside objects

Allocator metadata
an **overflow** corrupts it; a forged value can become an arbitrary write

Function pointer
overwriting it redirects the next indirect call

C++ vtable pointer
overwriting it redirects the next virtual call

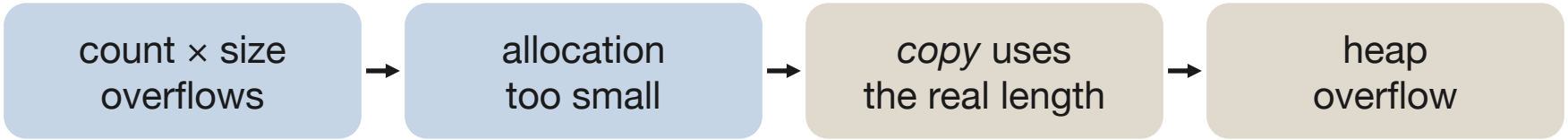
Use-after-free
a freed block is reused for attacker data behind a stale pointer

Double-free
the free list is corrupted, so later allocations alias

Type confusion
an object is used as the wrong type, so fields are misread

Integer Errors

An integer mistake becomes a memory bug when it produces the wrong size for an allocation or a copy.



Signed-to-unsigned mismatch: a negative length passes a signed upper-bound check, then becomes a huge unsigned size

Off-by-one: a write one position past can change a saved value or the low byte of a pointer, or begin a larger overflow

How does a wrapped size calculation turn into a memory overflow?

Defenses Come in Layers

No single defense solves all the problems.

Each addresses a specific attack, so they are deployed together.

Find and prevent bugs: **auditing, sanitizers, fuzzing, memory-safe languages**

Make a surviving bug harder to exploit: **non-executable memory, ASLR, stack canaries, hardened allocators**

Detect tampering at run time: **control-flow integrity, Intel CET, ARM Pointer Authentication, ARM Memory Tagging**

Layer 1: Find and Prevent Bugs

- **Auditing, static analysis, and compiler warnings** flag missing checks and unsafe calls before the code ships.
- **Sanitizers** add runtime checks that report memory and arithmetic errors in whatever code the tests exercise.
- **Fuzzing** feeds malformed input and watches for failures; coverage-guided fuzzers steer input toward new code paths.
- **Memory-safe languages** such as Rust and Go check bounds and manage lifetimes, so an invalid access is caught instead of corrupting memory.

The guarantee stops at unsafe code and at native libraries the program calls.

Non-Executable Memory

Marking data pages **non-executable** stops injected code from running and forces attackers to reuse code that's already there.

Three related terms:

NX
the hardware page permission: *this page is not executable*

DEP
the operating-system protection built on NX

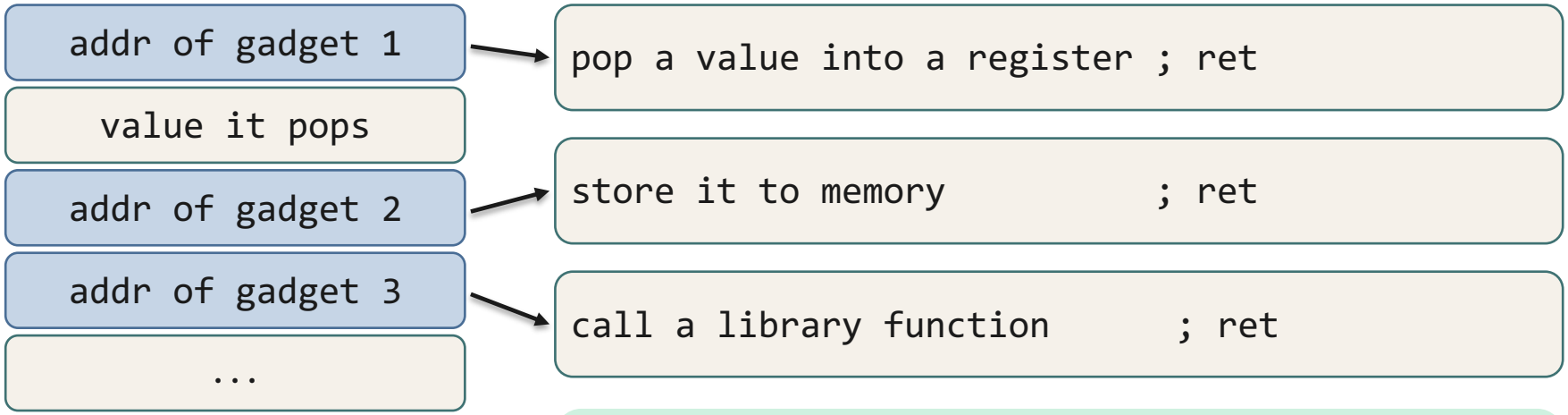
W^X
the policy: no page is writable and executable at once

It does not stop code reuse:
the attacker jumps to code that is already executable, which leads to return-to-libc and ROP attacks

Code Reuse: return-to-libc and ROP

Non-executable memory does not stop reuse of existing code.

- **return-to-libc** jumps to a library function
- Return-oriented-programming (ROP) chains fragments of existing code.



attacker-controlled stack

Each `ret` jumps to the next address on the stack, so gadgets run in sequence.
They can do arbitrary work, with no injected code.

Address Space Layout Randomization (ASLR)

Every reuse attack needs an address to aim at.
ASLR places code and data at unpredictable locations so the attacker cannot know it.

Strength is its **entropy**: the amount of randomness in the placement.

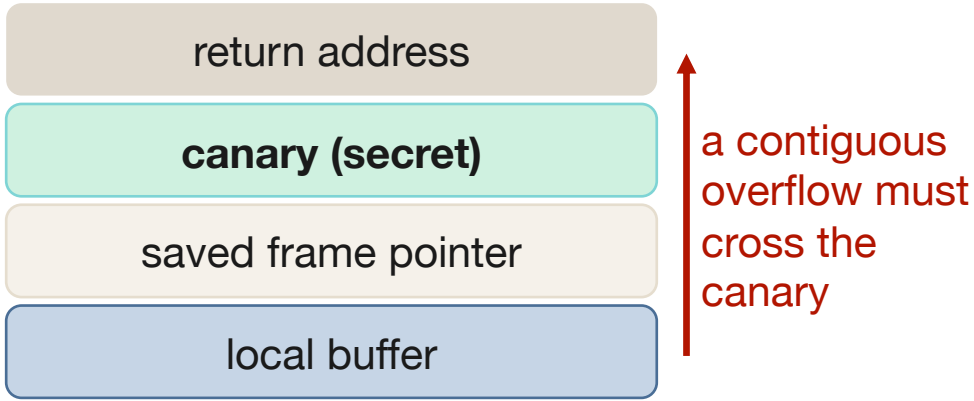
More entropy means more addresses to guess

A single leaked pointer exposes a region's base and undoes the randomization for that whole region.

Why does one leaked address undo randomization for an entire region?

Stack Canaries

A secret value sits between the local buffers and the return address and is checked before the function returns.



Catches a contiguous overflow that wrote through the canary on its way to the return address.

Misses a heap bug, a leaked canary that the attacker rewrites, and a data-only attack that never reaches it.

Hardened Allocators

The heap bugs we covered (overflow into metadata, use-after-free, and double-free) all abuse how the allocator stores and reuses blocks.

- A hardened allocator **checks and protects its own metadata**
  - A corrupted value is caught instead of trusted
- It **delays reuse of freed blocks**
  - This shrinks the window in which a stale pointer lands on attacker data.
- Relocates allocator data tracking
  - Some keep the bookkeeping **away from the user data** so an overflow out of a block cannot modify the lists of what is free and what is in use.
- It raises the complexity of a heap exploit
  - It does not make the heap memory-safe.

Control-Flow Integrity (CFI)

CFI checks, at run time, that an indirect call or jump lands on a target the program could legitimately reach — one whose function type matches the call.

An indirect transfer to a valid target of the right type is allowed.

A jump into the middle of a function or to a gadget is rejected.

A target that is wrong but still inside the allowed set still passes.

Forward-edge checks do not protect returns.

Used in Windows, in Clang-built software including Android and Chrome, and the Linux kernel

Apple's systems reach the same goal mainly through hardware Pointer Authentication.

Hardware-Assisted Defenses

Recent processors add checks in hardware that software alone could not make cheap enough to always run.

Intel CET

Control-flow Enforcement Technology

A hardware **shadow stack** keeps a protected copy of each return address and compares it with the one on the normal stack.

Indirect branch tracking requires indirect calls and jumps to land on marked targets.

ARM PAC

Pointer Authentication Code

Pointer Authentication adds a **check value**, computed with a secret key held in the processor, to selected pointers and verifies it before use, so tampering is detected.

The check value sits in the pointer's unused high bits, so a protected pointer is no larger than an ordinary one.

ARM MTE

Memory Tagging Extension

Memory Tagging gives each small block of memory a **tag** and stores a **matching tag** in the pointer.

An access whose tags disagree faults, catching many overflows and use-after-free.

Defense in Depth … and Its Gap

Independent defenses stacked together force an exploit to defeat every defense on its path, pushing attackers toward rarer, less reliable conditions.

Each layer we reviewed answers a specific attack and leaves something open.
Combined, they raise the cost of the whole chain.

A data-only attack changes a value without diverting control flow, so it slips past most of these.
A memory-safe language is the stronger fix.

Which defenses does a data-only attack bypass, and why?

Part 2
Format-String Bugs

What a Format String Is

The `printf` family builds its output from a format string whose directives, starting with %, each consume one argument.

```
printf("%s scored %d\n", name, score);
```

`%d`	an integer
`%s`	a string (the argument is a pointer to it)
`%x`	a value in hexadecimal

The Bug

The program passes untrusted input as the format string itself.

```
printf(user_input);
```

instead of

```
printf("%s", user_input);
```

In the first line, any % in the user's input is read as a directive.
In the second, the input is only ever data.

What Goes Wrong: Leaking Memory

With more directives than the call supplied arguments, the function keeps reading values that were never provided.

```
input:  %x %x %x %x
```

→

```
f7a1b2c0 00000000 7ffe4a10 08048430
```

sample output — values from the stack

Those values come from the stack and registers where arguments would have been. A leaked address among them exposes a region's base and defeats ASLR — the same connection from the review.

Reading Further, and Writing

%s treats the next value as a pointer and prints what it points to, often disclosing more data, or crashing if the value is not a valid address.

%n does the opposite: it **writes** a number back into memory.
So, a format-string bug is not only a *read* — it can be a *write*.

A single trusted-input mistake yields both a read and a write primitive.

Why It Belongs This Week … and the Fix

No overflow is involved.
A function trusted its input and handed attacker data to a function that treats it as instructions.

Fix: never pass untrusted data as the format string.
Pass a constant format and the data as an argument.

```
printf("%s", user_input);
```

Compilers warn about a non-constant format string (`-Wformat-security`), and hardened builds add runtime checks.

One Rule for the Whole Week: Assumptions are Bad

Nearly every bug this week came from a program that trusted input it should have checked: whether the input was too long, the wrong size, or a format string.

The defenses raise the cost of getting it wrong.

Checking all input, watching out for overflows/underflows, and using a memory-safe language where you can, removes the bug at the source.

The end.