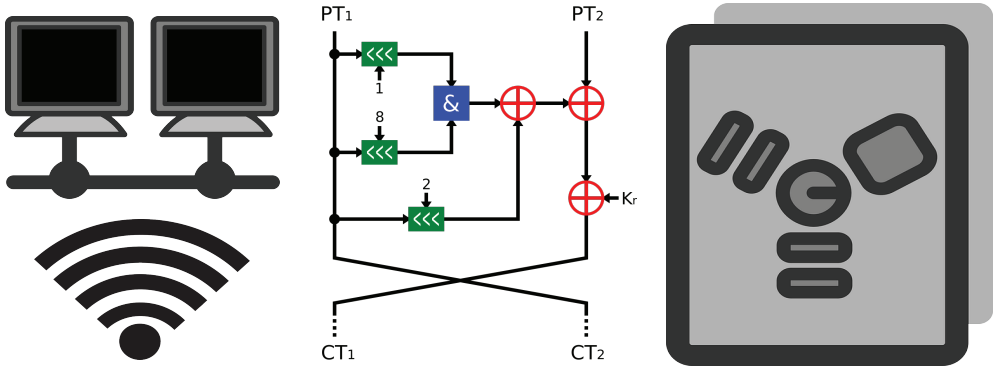
CSE 40567 / 60567: Computer Security



Software Security 6

Homework #5 has been released. It is due 4/2 at 11:59PM

See **Assignments Page** on the course website for details

Protection Mechanisms Against Attack

Basic Strategies

- Check if known values in memory are being overwritten
- Have the compiler perform bounds checking
- Make it difficult for an attacker to find the necessary memory offsets
- Explicitly mark regions of memory as being non-executable

Philosophical question: can't we solve this with good coding practices?

Canaries

Goal: detect that a return address has been altered before a function returns

Implementation: place a known value between a buffer and control data on the stack

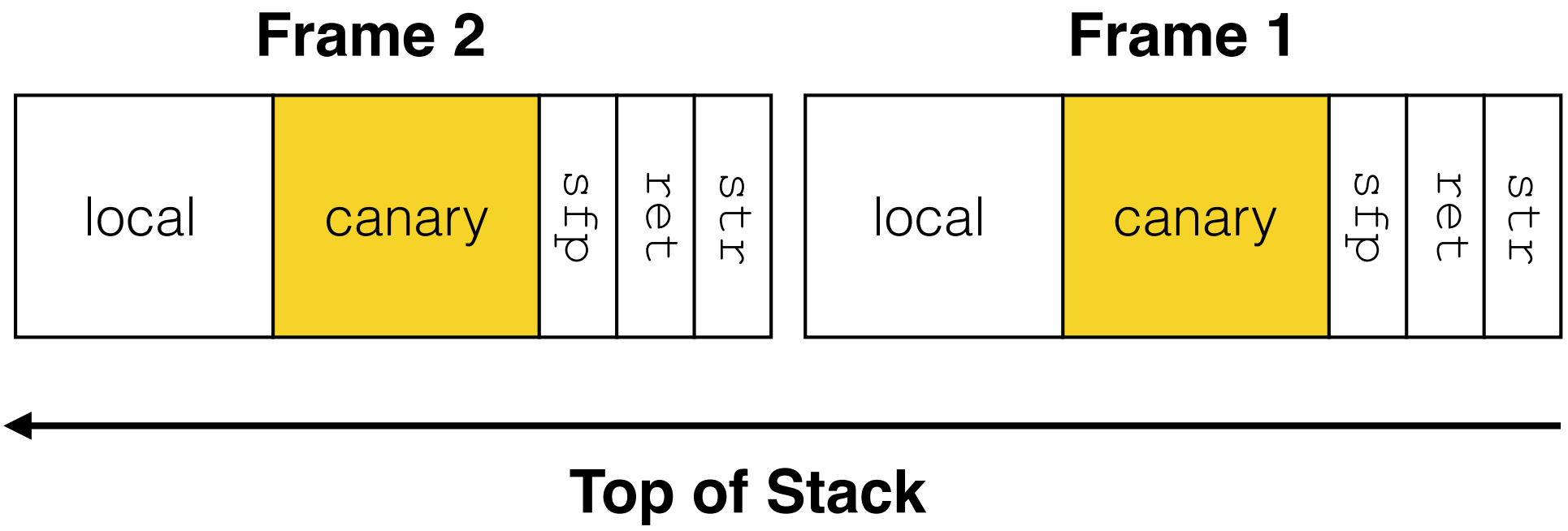
Three basic strategies:

1. Terminator canaries
2. Random canaries
3. XOR canaries

Canary in a Coal Mine

https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf

Canary word next to return address



Random canaries

- Random canary string chosen at program invocation
- Canary string is inserted into every stack frame
- To bypass the canary, the attacker needs to know what it is ahead of time

Operation:

1. Canary is verified before returning from function
2. If canary changed, exit program ⟵ Potential for DoS attack

Terminator canaries

Use a known canary: `{0, newline, linefeed, EOF}`

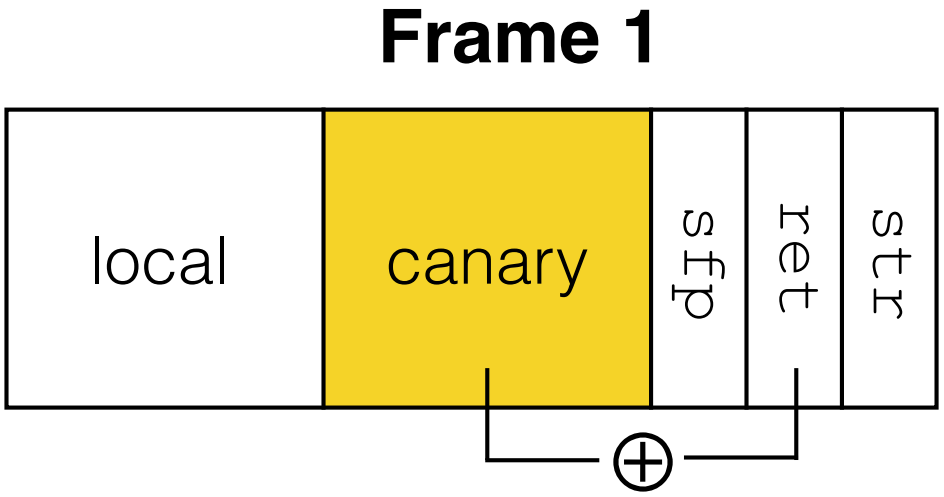
How do these characters help us?

- String functions will not copy beyond a terminator
  - e.g., `strcpy()` returns when copying a null character
- Attacker can't use string functions to corrupt the stack

Potential pitfall: the canary is known to the attacker

Random `XOR` canaries

Variation on the theme of a random canary: also consider control portions of the stack



Check that incorporates canary and `ret`

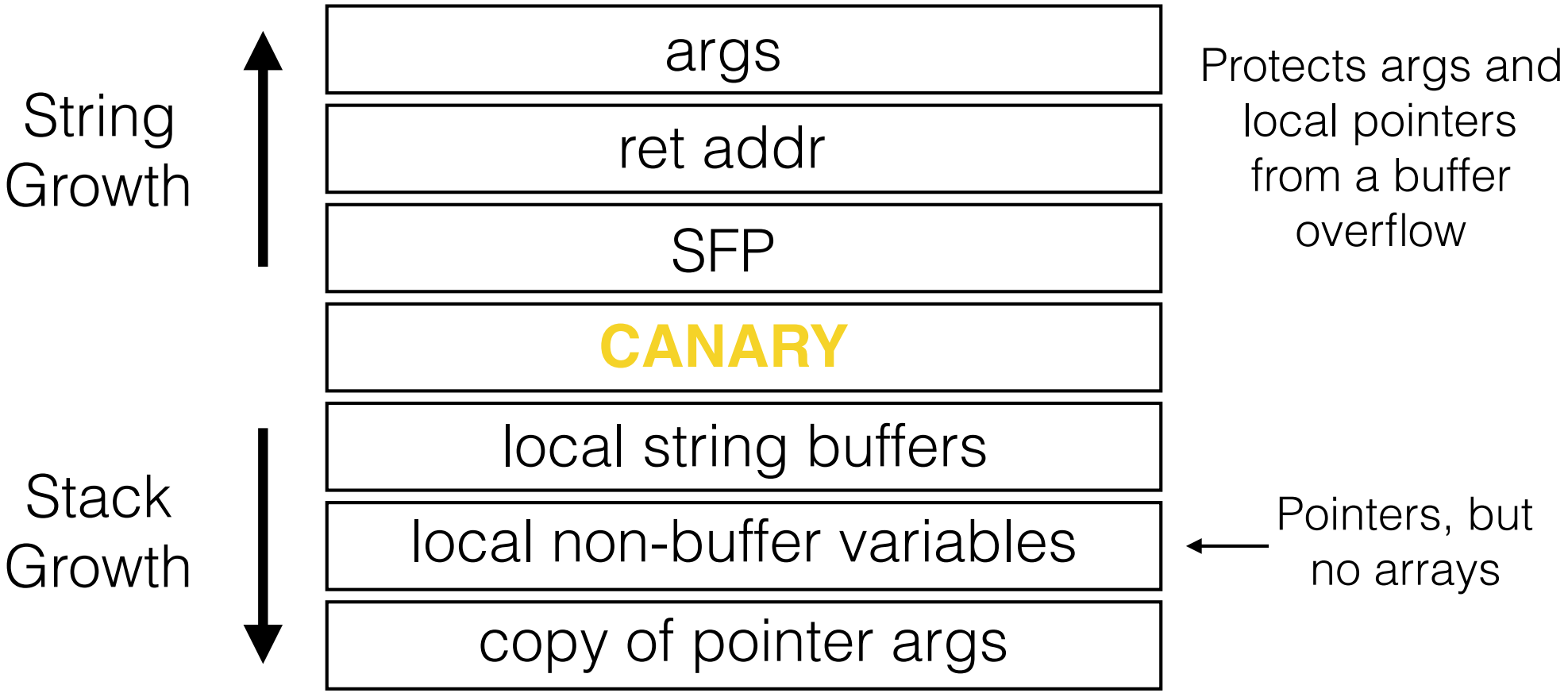
To attack, a matching `xor` result must be computed from a new canary and `ret` address. This requires knowledge of the original canary and canary algorithm.

Stackguard

- The three canary strategies are implemented as a patch to `gcc`
- Minimal impact on performance
  - Apache: 8% slower
- Caveat: canaries are not fool-proof
  - Some buffer overflow attacks will leave the canaries intact

ProPolice (IBM)

- GCC 4.1: `-fstack-protector` and `-fstack-protector-all`
- Re-arrange stack layout to prevent pointer overflow



`-fstack-protector-strong` (Google)

- Attempt to balance security and performance
- Protects more functions than `-fstack-protector`
  - Which protects less than 2% of functions
- But not as many as `-fstack-protector-all`
  - Which protects everything, whether needed or not

Standard for some OSs: OpenBSD, Hardened Gentoo

Optional for others: Debian, FreeBSD

Visual Studio's `/GS` switch

Default compiler option since VS 2005

Added to all functions in VS 2010

Combination of ProPolice and random canary

If there is a "cookie" mismatch, the default behavior is to call `_exit(3)`

`___security_cookie` = canary

Function prolog and epilog

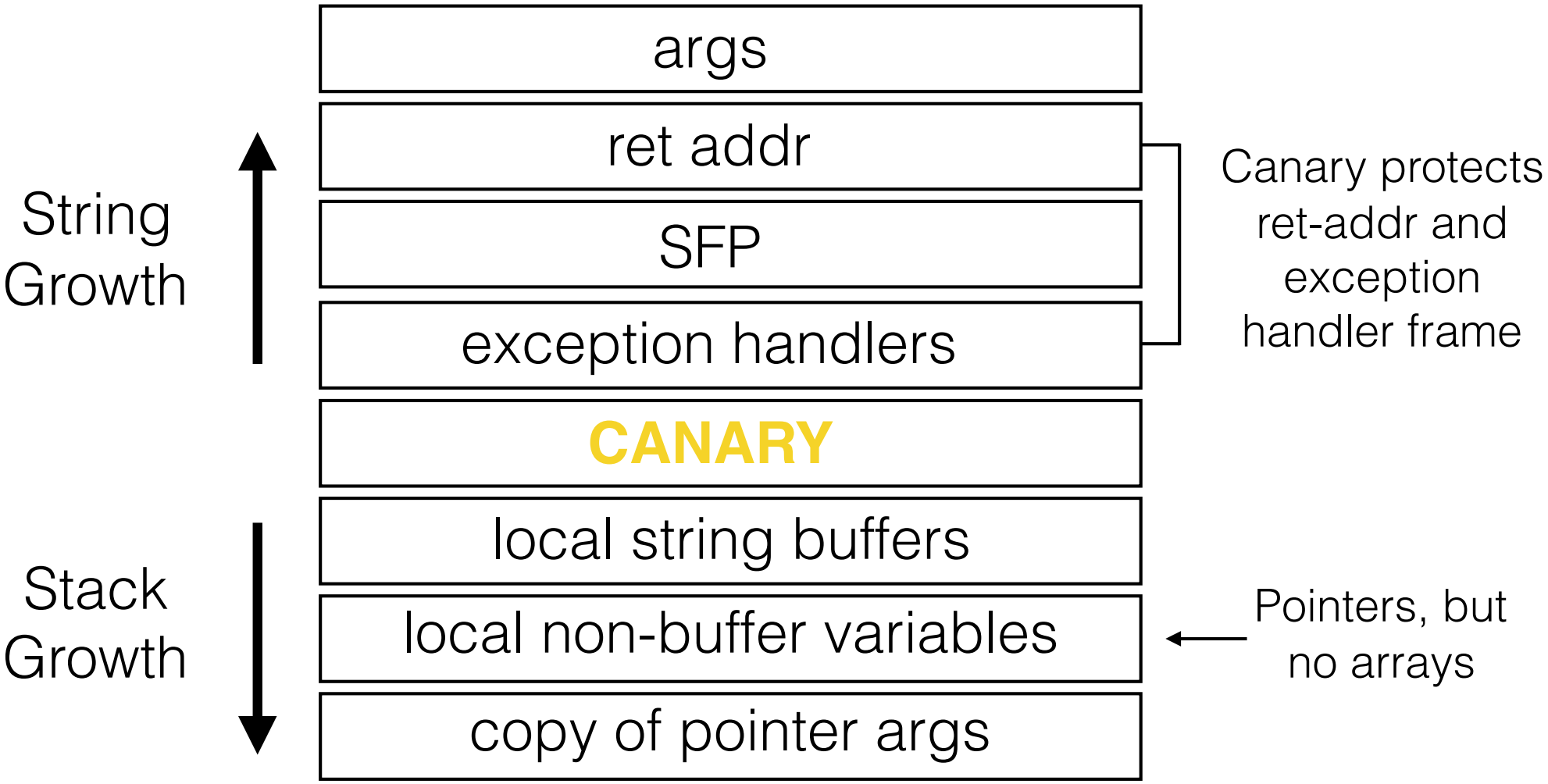
FuncFon prolog:

```
sub esp, 8          // allocate 8 bytes for cookie
mov eax, DWORD PTR ___security_cookie
xor eax, esp       // xor cookie with current esp
mov DWORD PTR [esp+8], eax   // save in stack
```

FuncFon epilog:

```
mov ecx, DWORD PTR [esp+8]
xor ecx, esp
call @__security_check_cookie@4
add esp, 8
```

/GS stack frame



/GS is not foolproof

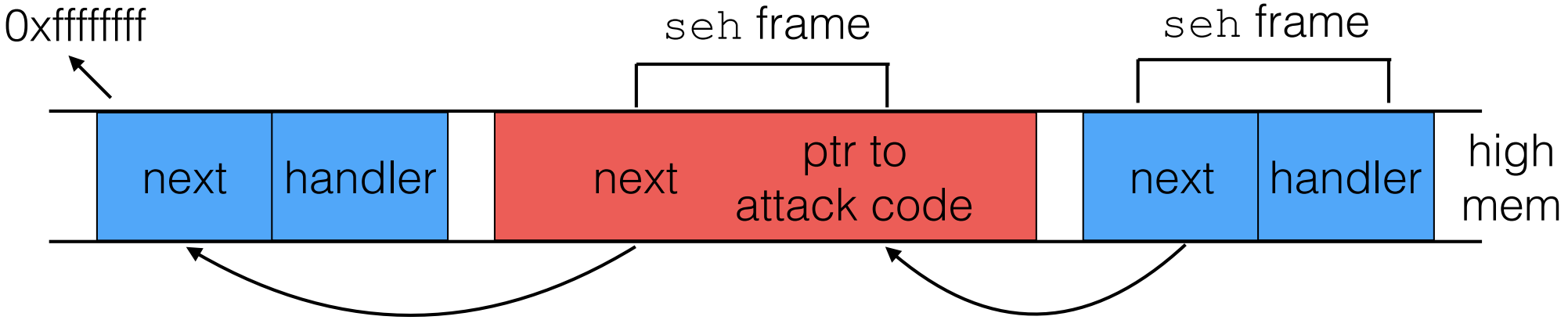
Evasion is possible: trigger exception before **canary** is checked

When an exception is thrown, the dispatcher walks up the exception list until the handler is found

- If no handler is found, the default is used

After overflow: handler points to attacker's code

exception triggered $\Longrightarrow$ control hijack



Additional VS flags

`/SAFESEH`: linker flag

- Linker produces a binary with a table of safe exception handlers
- System will not jump to an exception handler not on list

Additional VS flags

`/SEHOP`: platform defense (since Vista SP1)

- `SEH` attacks typically corrupt the “next” entry in the `SEH` list
- `/SEHOP` adds a dummy record at top of `SEH` list
- When exception occurs, the dispatcher walks up the list and verifies that the dummy record is there. If not, the process terminates.

If recompiling is not an option: Libsafe

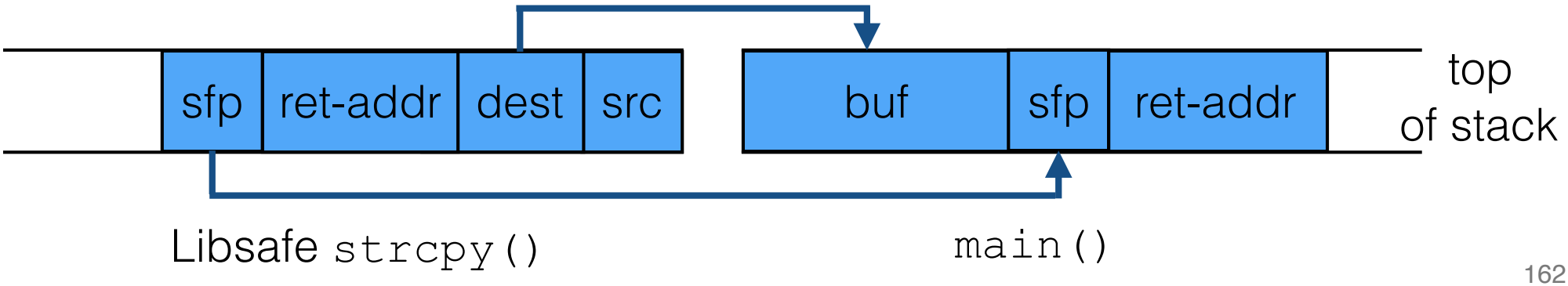
http://directory.fsf.org/wiki/Libsafe

- Dynamically loaded library
- Intercepts calls to `strcpy(dest, src)`

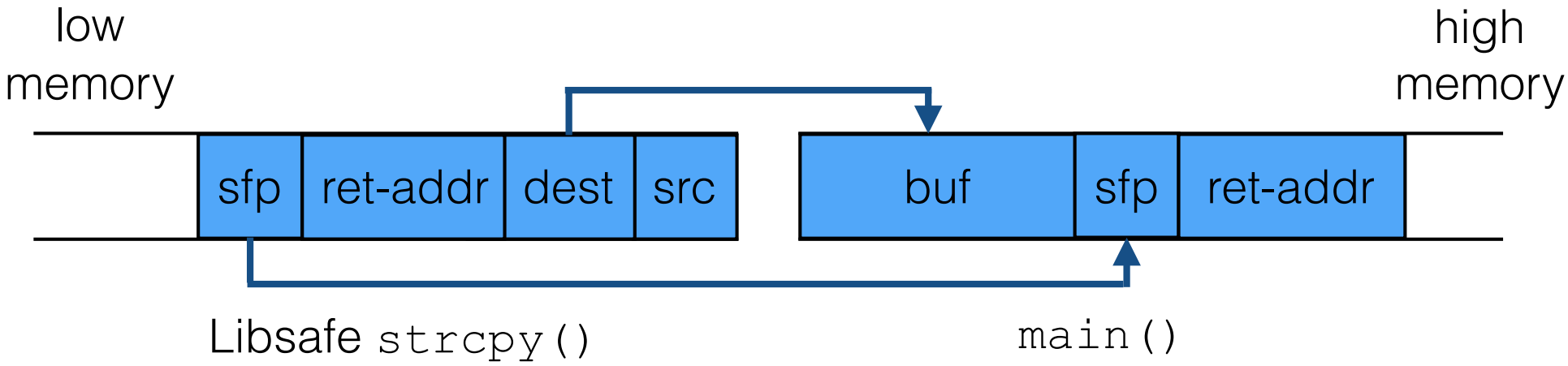
Validates sufficient space in current stack frame:

`|frame-pointer – dest| > strlen(src)`

If so, does `strcpy()`. Otherwise terminates application.



Libsafe is not foolproof



`strcpy()` can overwrite a pointer between buf and sfp.

Compiler-based bounds checking

Bounds Checking for C (http://www.doc.ic.ac.uk/~phjk/BoundsChecking.html)

- Every pointer expression derives a new pointer from a unique original pointer.
- Every pointer value is valid for just one allocated storage region.
- Pointer arithmetic expressions are checked for validity
- Implemented as patch to gcc

Compiler-based bounds checking

SAFECode (http://safecode.cs.illinois.edu/)

- Array bounds checking
- Loads and stores only access valid memory objects
- Type safety for a subset of memory objects proven to be type-safe
- Sound operational semantics in the face of dangling pointer errors
- Optional dangling pointer detection (induces overhead)

Compiler-based bounds checking

AddressSanitizer (https://github.com/google/sanitizers)

- Compiler instrumentation module (currently, an LLVM pass)

Can find:

Use after free (dangling pointer dereference), Heap buffer overflow, Stack buffer overflow, Global buffer overflow, Use after return, Initialization order bugs, Memory leaks

More on heap protection in a moment…

Tagging

- Tag the type of data in memory, and perform strong type checks during program execution
- Two key tags:
    1. memory is non-executable
    2. memory is non-allocated

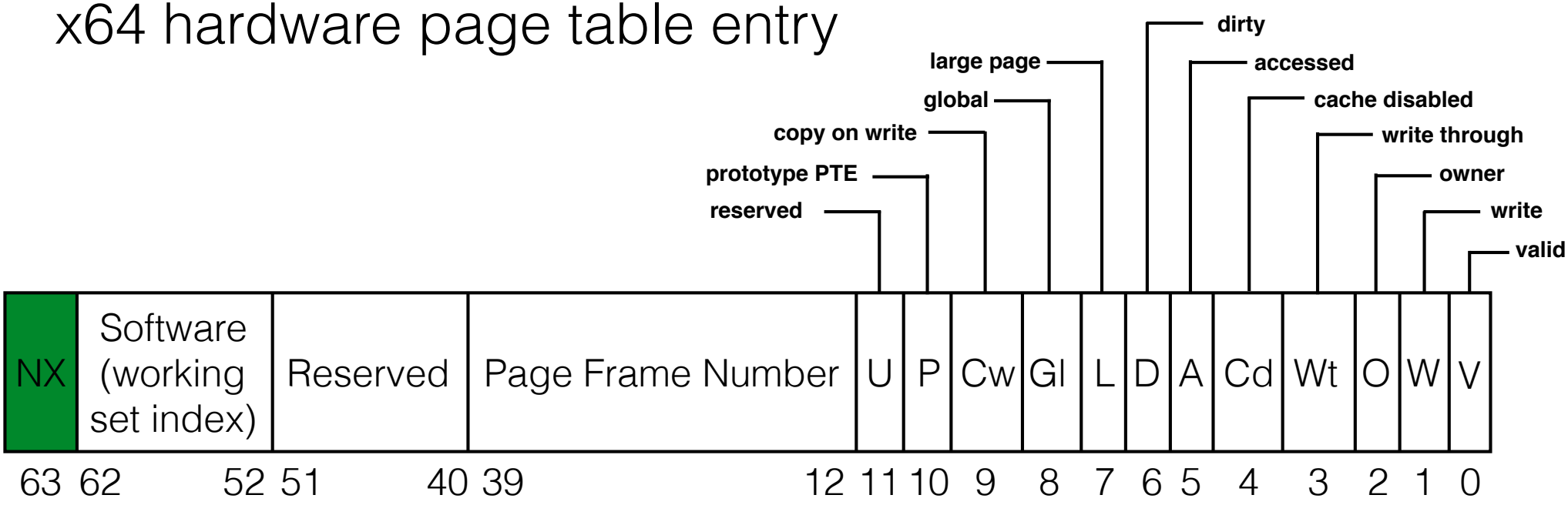
Implemented via software or **hardware** (far more difficult to thwart)

An Intel Core i7 2600K processor CC BY-SA 3.0 Eric Gaba

No eXecute (NX) bit

Supported by Intel, AMD and ARM processors

x64 hardware page table entry



W xor X (OpenBSD)

Many bugs are exploitable because the address space contains memory that is both writeable and executable (permissions = `W ∧ X`)

A serious hinderance would be to ensure no pages have `W ∧ X` permission

Implementation: use the NX bit to control permission at the hardware level on architectures that support it

As of 2015, userland and kernel are protected for AMD64

Address space layout randomization (ASLR)

- Randomize the memory space of a process in order to prevent an attacker from finding addresses or functions
  - Recall our discussions of why some buffer and heap overflow exploits are difficult to deploy in 2019
- Not always present in an OS (e.g., Windows)

http://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/

ASLR in Linux

`/proc/sys/kernel/randomize_va_space`

Mode 2 enabled by default

0 – No randomization. Everything is static.

1 – Conservative randomization. Shared libraries, stack, `mmap()`, VDSO and heap are randomized.

2 – Full randomization. In addition to elements listed in the previous point, memory managed through `brk()` is also randomized.

How effective is ASLR?

- Limitations on the amount of entropy
  - A 32-bit system provides less entropy than a 64-bit system
- Only partial protection is provided if binaries haven't been compiled as a Position Independent Executable (PIE)
  - Even with ASLR mode 2, overflow attacks are still possible if executable or library has not been compiled with explicit protection

https://media.blackhat.com/bh-us-10/whitepapers/Le/BlackHat-USA-2010-Le-Paper-Payload-already-inside-data-reuse-for-ROP-exploits-wp.pdf

Example of an executable that is not randomized despite `randomize_va_space = 2`

```
#include <stdlib.h>
#include <stdio.h>

void* getEIP () {
   return __builtin_return_address(0)-0x5;
};

int main(int argc, char** argv){
   printf("EBP located at: %p\n",getEIP());
   return 0;
}
```

ASLR enabled, default compilation flags

Libraries located at random locations:

```
$ ldd ./foo
   linux-vdso.so.1 =>  (0x00007ffe088d2000)
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
                (0x00007f2ab1fdb000)
   /lib64/ld-linux-x86-64.so.2 (0x00007f2ab23b9000)
$ ldd ./foo
   linux-vdso.so.1 =>  (0x00007ffe565c3000)
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
                (0x00007f17f6b28000)
   /lib64/ld-linux-x86-64.so.2 (0x00007f17f6f06000)
```

```
$ ./foo
EBP located at: 0x400516
$ ./foo
EBP located at: 0x400516
$ ./foo
EBP located at: 0x400516
```

`.text` section is in the same place!

ASLR enabled, `gcc -fPIE -pie`

Libraries located at random locations:

```
$ ldd ./foo
   linux-vdso.so.1 =>  (0x00007ffe1b6c6000)
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
                (0x00007fb2cbb48000)
   /lib64/ld-linux-x86-64.so.2 (0x00007fb2cc128000)
$ ldd ./foo
   linux-vdso.so.1 =>  (0x00007fff095d0000)
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
                (0x00007fe7bd023000)
   /lib64/ld-linux-x86-64.so.2 (0x00007fe7bd603000)
```

```
$ ./foo
EBP located at: 0x7fb225eb474e
$ ./foo
EBP located at: 0x7f981c4bc74e
$ ./foo
EBP located at: 0x7fd252c4d74e
```

`.text` section is now randomized

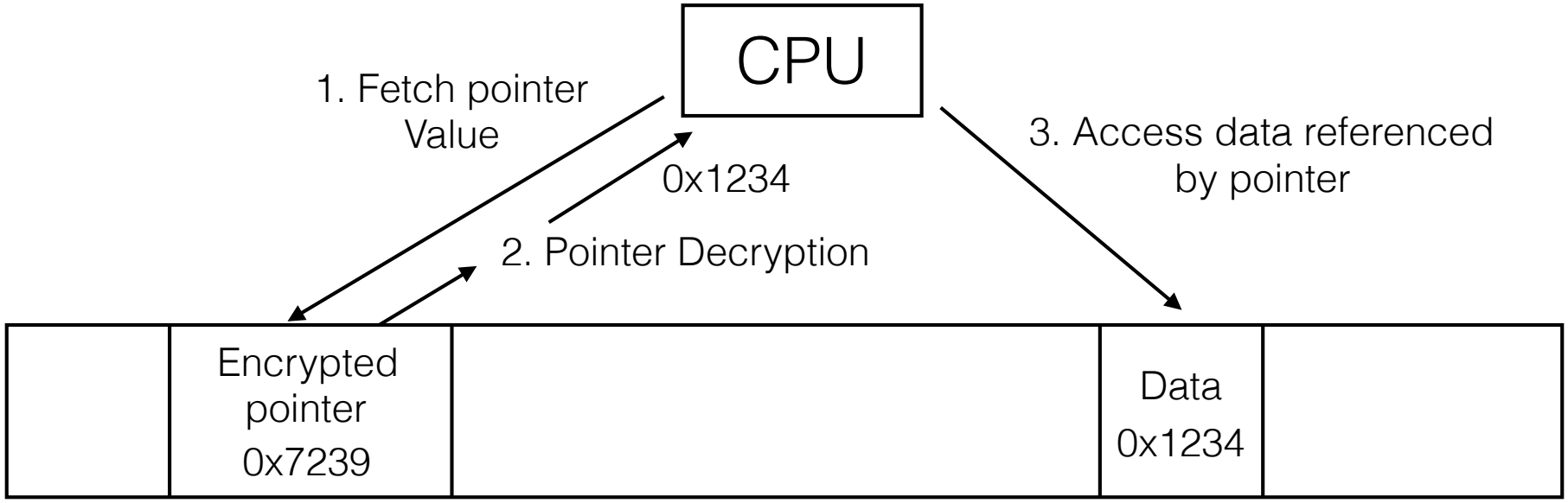
Protecting the heap

- W xor X works for the heap as well
  - But some applications need an executable heap (e.g., JIT programs)
- **Pointguard**
- **AddressSanitizer**

Pointguard

Protects function pointers and `setjmp` buffers by encrypting them.

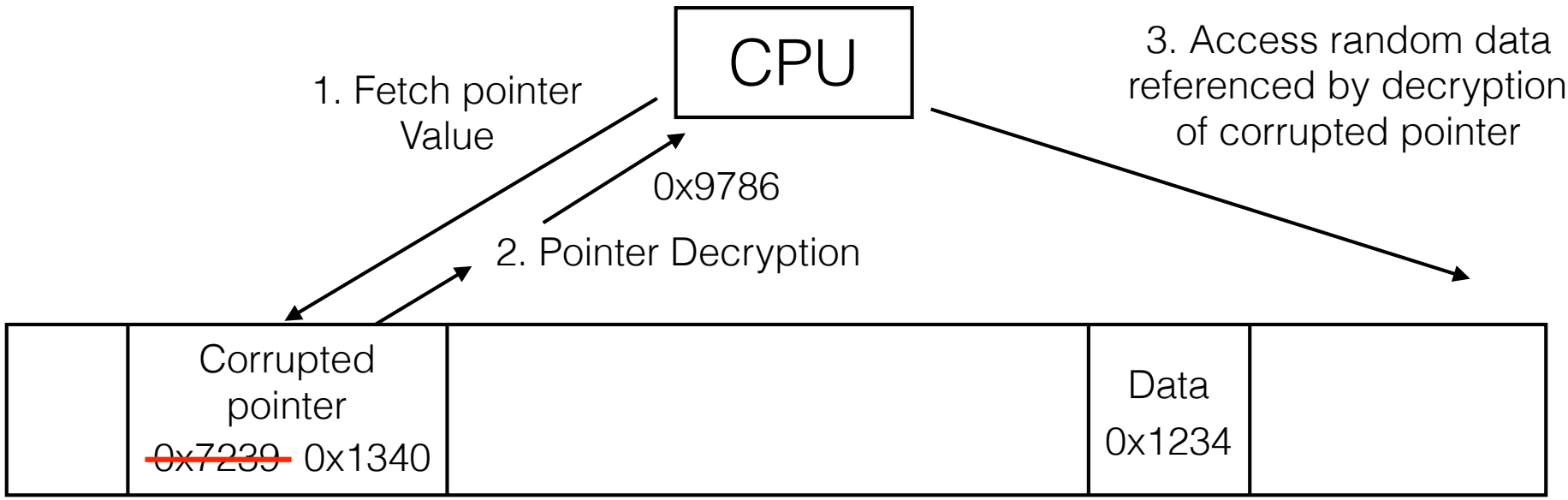
Pointguard Pointer Dereference



https://www.usenix.org/legacy/event/sec03/tech/full_papers/cowan/cowan.pdf

Pointguard

Pointguard Pointer Dereference Under Attack



Address sanitizer

- Run-time library which replaces the `malloc()` function.
- Compile and link program using `clang` with the `-fsanitize=address` switch.
- To get a reasonable performance add `-O1` or higher.
  - Increases processing time by 73% and memory usage by 340%
- To get nicer stack traces in error messages add `-fno-omit-frame-pointer`

https://github.com/google/sanitizers/wiki/AddressSanitizer

Address sanitizer

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array[0] = 0;
  int res = array[argc + 100];  // overflow
  delete [] array;
  return res;
}
```

```
==6226== ERROR: AddressSanitizer: heap-buffer-overflow on address
0x603e0001fdf4 at pc 0x417f8c bp 0x7fff64c0c010 sp 0x7fff64c0c008
READ of size 4 at 0x603e0001fdf4 thread T0
    #0 0x417f8b in main example_HeapOutOfBounds.cc:5
    #1 0x7fa97c09376c (/lib/x86_64-linux-gnu/libc.so.6+0x2176c)
    #2 0x417e54 (a.out+0x417e54)
0x603e0001fdf4 is located 4 bytes to the right of 400-byte region
[0x603e0001fc60,0x603e0001fdf0)
allocated by thread T0 here:
    #0 0x40d312 in operator new[](unsigned long) /home/kcc/llvm/
projects/compiler-rt/lib/asan/asan_new_delete.cc:46
    #1 0x417f1c in main example_HeapOutOfBounds.cc:3
```

https://github.com/google/sanitizers/wiki/AddressSanitizerExampleHeapOutOfBounds

Mitigating integer overflows

- Sub-typing: define rules to express relationships between types, which can be used to validate the safety of a program
  - http://web.archive.org/web/20121010025025/http://www.cs.cmu.edu/~dbrumley/pubs/integer-ndss-07.pdf
- As-if infinitely ranged integers: does not break or inhibit existing optimizations
  - http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=9019
- **Easy solution:** use a language with arbitrary-precision arithmetic and type safety
  - e.g., python

Defenses are not perfect

- We've seen some good attacks, counter-measures, and counter-counter measures…
  - Security reduces to an arms race
  - Constraints of the von Neumann architecture

Your best strategy: avoid problematic function calls and audit code before deployment