CSci 4271W
Development of Secure Software Systems
Day 7: Memory corruption 3, mitigation

Stephen McCamant (he/him)
University of Minnesota, Computer Science & Engineering

Memory corruption

- Memory corruption bugs happen when a program writes data to an area of memory that it shouldn't.
- Type-safe languages such as Java, OCaml, Rust, Swift, and Go can prevent most such bugs.
- Mitigation 1: use a type-safe language for development.

Are we done?

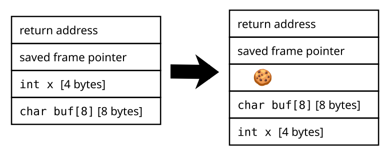
Some code still needs to run with substantial C/C++ code bases. What can we do?

- Development: Lint/static analysis (SAST), compiler warnings, code review
- Compiler: Stack protector, FORTIFY, ASAN, CFI
- OS: W⊕X/DEP, ASLR, Isolation/sandboxing
- Processor: ARMv8 PAC

Stack protector

GCC and Clang have `-fstack-protector` on by default.

- Stack cookies in all functions with stack buffers
- Buffers moved to "top" of local variables



Shadow stack

- The stack cookie value needs to be stored somewhere safe
  - So, why not store all return addresses somewhere safe?
- Needs to be a stack, but separate from the one where buffers go
- Supported by Clang for AArch64 (including Android) and RISC-V

FORTIFY_SOURCE

- GCC and Clang have the `-D_FORTIFY_SOURCE` option
- Protects memcpy, strcpy, strcat, sprintf into static buffers.

```
char buf[2];
strcpy(buf, "abc"); //compile-time warning!
```

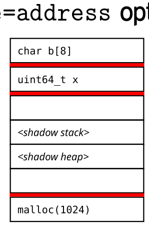
```
char *p = "01234567"
char buf[8];
strcpy(buf, p); //run-time abort
```

```
char *p = "01234567"
char *buf = malloc(8)
strcpy(buf, p); // won't help here, alas
```

Address Sanitizer

GCC and Clang have the `-fsanitize=address` option

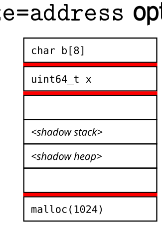
- All allocations (stack+heap) have "red zone" buffers
- Separate "shadow" memory records allocated regions
- All loads/stores checked against shadow records



Address Sanitizer

GCC and Clang have the `-fsanitize=address` option

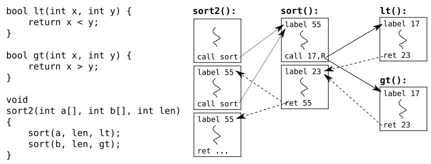
- All allocations (stack+heap) have "red zone" buffers
- Separate "shadow" memory records allocated regions
- All loads/stores checked against shadow records



Mostly a testing tool, because of high overhead

Control-flow integrity (CFI)

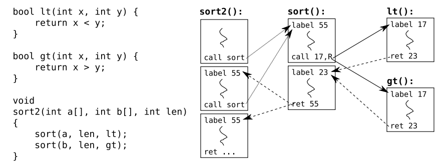
CFI checks that (indirect) calls only go to function start, returns only jump to after call sites.



Originally introduced by Abadi et al. in CCS 2005 (source for figures)

Control-flow integrity (CFI)

CFI checks that (indirect) calls only go to function start, returns only jump to after call sites.



Clang: `-fsanitize=cfi`
MSVC: `/guard:cf` (for calls)

CFI rewriting examples (Intel 32-bit)

```
call  [ebx+8]  ; call fptr   →   mov  eax, [ebx+8]        ; load fptr
                                 cmp  [eax+4], 12345678h  ; comp w/ID
                                 jne  error_label         ; if != fail
                                 call eax                 ; call fptr
                                 prefetchna [AABBCCDDh]   ; label ID

ret   10h      ; return      →   mov  ecx, [esp]    ; load ret
                                 add  esp, 14h      ; pop 20
                                 cmp  [ecx+4],      ; compare
                                      AABBCCDDh     ;     w/ID
                                 jne  error_label   ; if!=fail
                                 jmp  ecx           ; jump ret
```

CFI limitations 1

```
int main(int argc, char **argv) {
  int bad_idea = 0;
  char overflow_me[8];
  char *p = argv[1];
  while (*overflow_me++ = *p++);
  if (bad_idea)
    system("/bin/sh");
  return 0;
}
```

CFI can't stop overflows that don't change control flow

CFI limitations 2

```
void stooge() {
  char value[16];
  char ind[16];
  intptr_t index = 0;
  fgets(ind,15,stdin);
  index = strtol(ind,NULL,16);
  fgets(value+index,9,stdin);
  return; }
int am_i_root() {
  return geteuid() == 0; }
```

```
void harmless() {
  stooge();
  return; }
void why() {
  if (am_i_root()) return;
  stooge();
  system("/bin/sh"); }
int main(...) {
  if (am_i_root()) harmless();
  else why(); }
```

Standard CFI doesn't prevent returning to unintended but legitimate call sites.

OS: non-executable stack

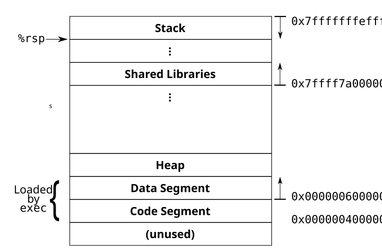
- Memory pages can be marked as writeable or an executable, but not both at once (W xor X, or DEP)
- This prevents jumping to code placed on the stack or heap.
  - But many library/system calls can load a new binary/shell (`exec`, `system`, `popen`) and libc is always in memory
- A "return to libc" attack works by overwriting the return address with a pointer to such a function

OS: non-executable stack

- Memory pages can be marked as writeable or an executable, but not both at once (W xor X, or DEP)
- This prevents jumping to code placed on the stack or heap.
  - But many library/system calls can load a new binary/shell (`exec`, `system`, `popen`) and libc is always in memory
- A "return to libc" attack works by overwriting the return address with a pointer to such a function
- More general: "return-oriented programming" (ROP)

Address space...

Address space layout of a typical Linux/x86-64 process:



…layout randomization

Address space layout randomization (ASLR) randomizes:

- Stack location (always): hard to find the right address on stack to jump to
- Heap location (often): hard to find address of heap buffer to stash shellcode
- Shared libraries (often): hard to find address of libc
- Code/data segments (sometimes): hard to find address of existing code

ASLR problems

- 32-bit addresses are easy(ish) to guess (w/big NOP sled)
- Legacy code can prevent relocating libraries/code segment
- Relative offsets are maintained (for ret2libc/ROP)
- Linux default does not relocate code/data segments
- Uninitialized read, format string, interpreter bugs can leak secrets (ASLR offsets, also cookies)

Hardware: PAC

- "64-bit" architectures don't actually use all the bits in an address (e.g., 48 bits on x86-64, ARM-64)
- ARMv8 idea: use top 3–24 bits of code pointers to hold a "Pointer Authentication Code" (PAC).
- Processor using PACs has instructions to set a code, and check a code before jumping there.
- Each PAC is specific to a program context and a key. Used in recent versions of iOS/macOS.

Spot the bug(s)

```
void checkpassword(FILE *pwfile) {
  int taunt = 1;
  char password[10], input[10];
  char *inp = input;
  fgets(password,9,pwfile);
  password[8]='\0';
  printf("Enter password (at most 8 letters):");
  do {
    *inp = getchar();
  } while (*inp++ != '\n');
  input[8] = '\0';
  if (strncmp(input,password,8) == 0) taunt = 0;
  if (taunt) {
    printf("Loser, the password is definitely not ");
    printf(input);
  } else return success();
}
```