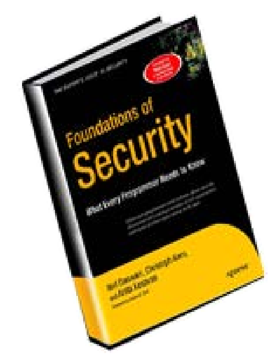
Chapter 5

Buffer Overflows



Agenda

- Buffer overflows: attacker hijacks machine
  - Attacker injects malicious code into program
- Preventable, but common
  (50% CERT advisories
  decade after Morris Worm)

- Fixes: Safe string libraries,
  StackGuard, Static Analysis

- Other Types of Overflows: Heap, Integer, …

6.1. Anatomy of a Buffer Overflow

- *Buffer*: memory used to store user input, has fixed maximum size

- *Buffer overflow*: when user input exceeds max buffer size
  - Extra input goes into unexpected memory locations

6.1.1. A Small Example

- Malicious user enters > 1024 chars, but `buf` can only store 1024 chars; extra chars overflow buffer

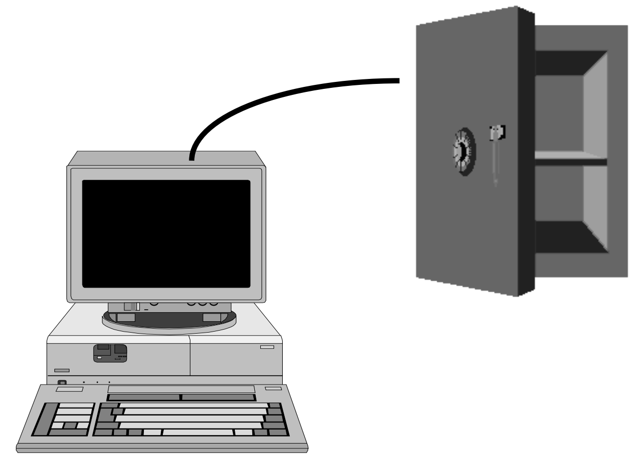
```
void get_input() {
    char buf[1024];
    gets(buf);
}
void main(int argc, char*argv[]){
    get_input();
}
```

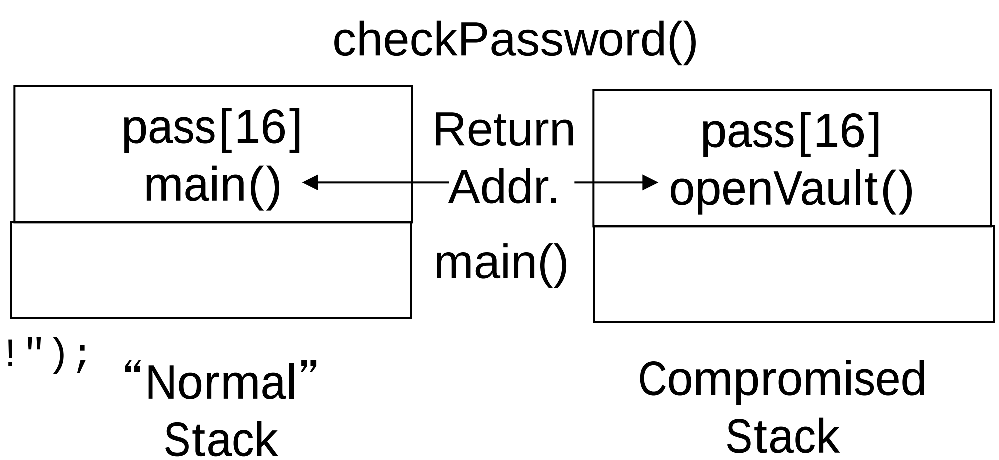
6.1.2. A More Detailed Example

```
int checkPassword() {
    char pass[16];
    bzero(pass, 16); // Initialize
    printf ("Enter password: ");
    gets(pass);
    if (strcmp(pass, "opensesame") == 0)
      return 1;
    else
      return 0;
}

void openVault() {
      // Opens the vault
}

main() {
     if (checkPassword()) {
         openVault();
         printf ("Vault opened!");
     }
}
```





6.1.2. `checkPassword()` Bugs

- *Execution stack*: maintains current function state and address of return function
- *Stack frame*: holds vars and data for function
- Extra user input (> 16 chars) overwrites return address
  - *Attack string*: 17-20th chars can specify address of `openVault()` to bypass check
  - Address can be found with source code or binary

6.1.2. Non-Executable Stacks Don't Solve It All

- Attack could overwrite return address to point to newly injected code

- NX stacks can prevent this, but not the vault example (jumping to an existing function)

- *Return-into-libc attack*: jump to library functions
  - e.g. `/bin/sh` or `cmd.exe` to gain access to a command shell (*shellcode*) and complete control

6.1.3. The safe_gets() Function

```
#define EOLN '\n'
void safe_gets (char *input, int max_chars) {
    if ((input == NULL) || (max_chars < 1))) return;
    if (max_chars == 1) { input[0] = 0; return; }
    int count = 0;
    char next_char;
    do {
        next_char = getchar(); // one character at a time
        if (next_char != EOLN)
        input[count++] = next_char;
    } while ((count < max_chars-1) && // leave space for null
             (next_char != EOLN));
    input[count]=0;
}
```

- Unlike `gets()`, takes parameter specifying max chars to insert in buffer
- Use in `checkPassword()` instead of `gets()` to eliminate buffer overflow vulnerability

```
5 safe_gets(pass, 16);
```

6.2. Safe String Libraries

- C – Avoid (no bounds checks): `strcpy()`, `strcat()`, `sprintf()`, `scanf()`
- Use safer versions (with bounds checking): `strncpy()`, `strncat()`, `fgets()`
- Microsoft's *StrSafe*, Messier and Viega's *SafeStr* do bounds checks, null termination
- Must pass the right buffer size to functions!
- C++: STL string class handles allocation
- Unlike compiled languages (C/C++), interpreted ones (Java/C#) enforce type safety, raise exceptions for buffer overflow

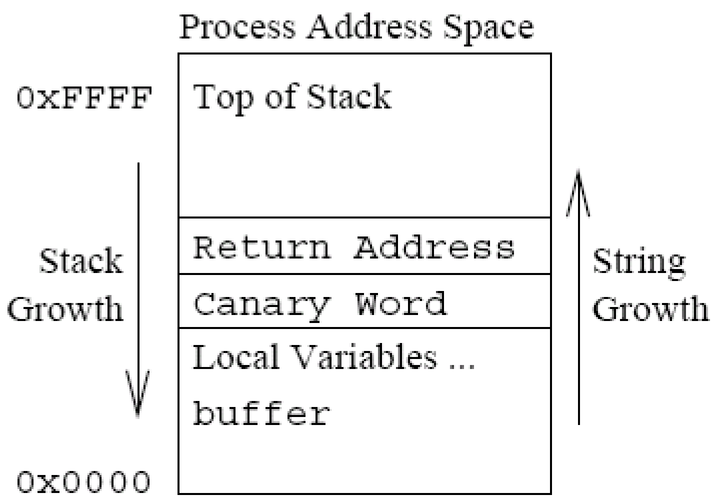
6.3. Additional Approaches

- Rewriting old string manipulation code is expensive, any other solutions?
- StackGuard/canaries (Crispin Cowan)
- Static checking (e.g. Coverity)
- Non-executable stacks
- Interpreted languages (e.g., Java, C#)

6.3.1. StackGuard

- *Canary*: random value, unpredictable to attacker
- Compiler technique: inserts *canary* before return address on stack

- Corrupt Canary: code halts program to thwart a possible attack



Source: C. Cowan et. al., StackGuard,

6.3.2. Static Analysis Tools

- *Static Analysis*: analyzing programs without running them
- Meta-level compilation
  - Find security, synchronization, and memory bugs
  - Detect frequent code patterns/idioms and flag code anomalies that don't fit
- Ex: Coverity, Fortify, Ounce Labs, Klockwork
  - Coverity found bugs in Linux device drivers

6.4. Performance

- Mitigating buffer overflow attacks incurs little performance cost
- Safe str functions take slightly longer to execute
- StackGuard canary adds small overhead
- But performance hit is negligible while security payoff is immense

6.5. Heap-Based Overflows

- Ex: `malloc()` in C provides a fix chunk of memory on the heap
- Unless `realloc()` called, attacker could also overflow heap buffer (fixed size), overwrite adjacent data to modify control path of program
- Same fixes: bounds-checking on input

6.6. Other Memory Corruption Vulnerabilities

- *Memory corruption vulnerability*: Attacker exploits programmer memory management error

- Other Examples
  - Format String Vulnerabilities
  - Integer Overflows
  - Used to launch many attacks including buffer overflow
  - Can crash program, take full control

6.6.1. Format String Vulnerabilities

- Format String in C directs how text is formatted for output: e.g. %d, %s
  - Can contain info on # chars (e.g. `%10s`)

```
void format_warning (char *buffer,
        char *username, char *message) {
    sprintf (buffer, "Warning: %10s -- %8s",
            message, username);
}
```

- If `message` or `username` greater than 10 or 8 chars, buffer overflows
  - attacker can input a username string to insert shellcode or desired return address

6.6.2. Integer Overflows (1)

- Exploits range of value integers can store
  - Ex: signed four-byte int stores between -2^{32} and $2^{32}-1$
  - Cause unexpected wrap-around scenarios

- Attacker passes `int` greater than max (positive) -> value wraps around to the min (negative!)
  - Can cause unexpected program behavior, possible buffer overflow exploits

6.6.2. Integer Overflows (2)

```
/* Writes str to buffer with offset
characters of blank spaces
preceding str. */
void formatStr(char *buffer, int buflen,
      int offset, char *str, int slen) {
      char message[slen+offset];
      int i;

      /* Write blank spaces */
      for (i = 0; i < offset; i++)
          message[i] = ' ';

      strncpy(message+offset, str, slen);
       // offset = 2^32!?
      strncpy(buffer, message, buflen);
      message[buflen-1] = 0;
       /*Null terminate*/
}
```

- Attacker sets `offset` = 2^{32}, wraps around to negative values!
  - write outside bounds of `message`
  - write arbitrary addresses on heap!

Summary

- Buffer overflows most common security threat!
  - Used in many worms such as Morris Worm
  - Affects both stacks and heaps
- Attacker can run desired code, hijack program
- Prevent by bounds-checking all buffers
  - And/or use StackGuard, Static Analysis…
- Type of Memory Corruption:
  - Format String Vulnerabilities, Integer Overflow, etc…