Buffer Overflow Countermeasures

Lecture 3

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Introduction

- countermeasures proposed and deployed in real-world systems and software
  - from hardware architecture, OS, compiler, library, to applications

Safer Functions

- to some memory copy functions, certain special characters decide whether the copy should end or not
  - *dangerous*: the length of data that can be copied is decided by data, which may be controlled by users
  - e.g., *strcpy*, *sprintf*, *strcat*, and *gets*
    - *strcpy:* https://www.cplusplus.com/reference/cstring/strcpy/
    - *sprintf:* https://www.cplusplus.com/reference/cstdio/sprintf/?kw=sprintf
    - *strcat:* https://www.cplusplus.com/reference/cstring/strcat/?kw=strcat
    - *gets:* https://www.cplusplus.com/reference/cstdio/gets/?kw=gets

Safer Functions (cont.)

- *safer* approach:
  - let developer have control: specifying the length in code
    - the size of target buffer decides the length, not the data
  - e.g., *strncpy*, *snprintf*, *strncat*, *fgets*
    - *strncpy*: https://www.cplusplus.com/reference/cstring/strncpy/?kw=strncpy
    - *snprintf*: https://www.cplusplus.com/reference/cstdio/snprintf/?kw=snprintf
    - *strncat*: https://www.cplusplus.com/reference/cstring/strncat/?kw=strncat
    - *fgets*: https://www.cplusplus.com/reference/cstdio/fgets/?kw=fgets
    - developers explicitly specify the max length of data to be copied into the target buffer
      - need to think about buffer size
  - relatively safer
    - what if developer intentionally specifies longer data?

Safer Dynamic Link Library

- drawback of safer function approach: require change in code
  - what if you only have binary?
    - difficult to change binary
- solution: dynamic link libraries (program uses library)
  - the library function code is not included in code's binary, instead, it is dynamically linked to code
  - safer library ⟹ safer code
- e.g., libsafe
  - perform boundary checking based on frame pointer
  - not allow copy beyond frame pointer
- e.g., libmib
  - support "limitless" strings, instead of fixed length string buffer
  - its own version functions like *strcpy*

Program Static Analyzer

- instead of eliminating buffer overflow, warns developers of buffer overflow vulnerabilities in code
  - implemented as command-line tool or in the editor
  - notify developers of unsafe code during developing phase
- e.g.,
  - ITS4 (C/C++)

Programing Language

- developer relies on programming language to develop program
- burden is removed if language does checking against buffer overflow
- e.g.,
  - Java and Python
    - automatic boundary checking
    - safer languages for avoiding buffer overflow

Other Approaches

- compiler: compile code, verify stack, and eliminate buffer overflow conditions
  - e.g., Stackshield and StackGuard:
    - check whether the return add. has been modified before a function returns
  - Stackshield
    - idea: save a copy of return addr. at safer place
      - at beginning of function, compiler inserts instructions to copy return addr. to a location that cannot be overflown
      - before returning from function, comparing return addr. on stack with the one in safer place
        - determine buffer overflow

Other Approaches

- compiler: compile code, verify stack, and eliminate buffer overflow conditions
  - e.g., Stackshield and StackGuard:
    - check whether the return add. has been modified before a function returns
  - StackGuard
    - idea: put a guard between return addr. and buffer
      - if return addr. is modified, the guard will also be modified
    - at the start of function,
      - the compiler adds a random value below return addr.
      - save a copy of random value at safer location (off stack)
    - before function returns
      - the canary is checked against the saved value

Other Approaches (cont.)

- operating system
  - loader program:
    - before execution, program is loaded into system
    - running environment is set up
  - dictate how the memory of program is laid out
  - e.g., Address Space Layout Randomization (ASLR)
    - randomize the layout of program memory, making it hard for attacker to guess memory address
- hardware architecture
  - modern CPU supports NX bit (No-eXecute)
    - separate code from data
  - OS marks certain memory areas as non-executable
    - processor refuses to execute any code residing in these areas
  - if stack is marked as non-executable

Address Randomization

- to succeed in buffer overflow, attackers need to get vulnerable program to "return" to their injected code
  - guess where the injected code will be
    - (easy) predict where the stack is located in memory
      - most OS places stack in fixed location
- necessary to place stack in fixed location? No.
- when compiler generates binary code from source code
  - add. of data are not hard-coded in binary code
  - instead, their addr. are calculated based on frame and stack pointers
    - add. of data are represented as the offset of these two pointers

Address Randomization

- for attackers, they need to know the absolute addr., not the offset
  - important: knowing the stack location
- idea to defend against buffer overflow?
  - randomize the start location of stack
    - benefits:
      - make attacker's job more hard
      - no effect on program
- Address Layout Randomization (ASLR)

Address Randomization on Linux

- to run program, OS loads program into system
  - set up stack and heap memory for program

- memory randomization is normally implemented in loader

- for Linux, ELF is common binary format for program
  - randomization can be carried out by ELF loader

Address Randomization on Linux

- e.g., simple program with two buffers: stack and heap
  - print out their add. to see whether stack and heap are allocated in different places every time we run program

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

void main() {
   char x[12];
   char *y = malloc (sizeof(char)*12);

   printf("address of buffer x (on stack): 0x%x\n", x);
   printf("address of buffer y (on heap): 0x%x\n", y);
}
```

Address Randomization on Linux

- e.g., simple program with two buffers: stack and heap
  - compile and run code under different randomization settings

```
// turn off randomization
$ sudo sysctl –w kernel.randomize_va_space=0
kernel.randomize_va_space =0
$ program.out
…
…
$ program.out
…
…
```

Address Randomization on Linux

- e.g., simple program with two buffers: stack and heap
  - compile and run code under different randomization settings

```
// randomize stack address
$ sudo sysctl –w kernel.randomize_va_space=1
kernel.randomize_va_space =1
$ program.out
…
…
$ program.out
…
…
```

Address Randomization on Linux

- e.g., simple program with two buffers: stack and heap
  - compile and run code under different randomization settings

```
// randomize heap address
$ sudo sysctl –w kernel.randomize_va_space=2
kernel.randomize_va_space =2
$ program.out
…
…
$ program.out
…
…
```

StackGuard

- stack-based buffer overflow attack needs to modify return add.
- solution to stack-based buffer overflow attack:
  - detect whether the return add. is modified before returning from a function
- StackGuard: place a guard between return add. and buffer, and use the guard to detect whether the return add. is modified
  - incorporated into compliers like gcc

StackGuard

- place non-predictable value (called guard) between buffer and return add.
  - before returning from function, check whether the value is modified
    - if modified, the return add. has been modified

detecting whether the return add. is overwritten

=

detecting whether the guard is modified