Secure Coding In C and C++

Cheng-shung Wang
NSC member in CCU Center
Outline

- What is the problem with C
- Common string manipulation errors
  - Mitigation strategies
- Detection and recovery
- Pointer subterfuge
  - Mitigation strategies
- Reference
What is the problem with C

- Portability
  - Problem arise from an imprecise understanding of the semantics of these logical abstractions and how they translate into machine-level instructions.
  - The C programming language is intended to be a lightweight language with small footprint.
  - When programmers fail to implement required logic because they assume it is handled by C (but it is not), it leads to vulnerabilities.
What is the problem with C

- Lack of type safety
  - Preservation
    - Preservation dictates that if a variable $x$ has type $t$ and $x$ evaluates to a value $v$, then $v$ also has type $t$.
  - Progress
    - Evaluation of an expression does not get stuck in any unexpected way.

- Legacy code
  - Some insecurity function such as `strcpy()` are standard, they continue to be supported and developers continue to use them.
Common string manipulation errors

- The four most common errors are
  - Unbounded string copies
  - Off-by-one errors
  - Null termination errors
  - String truncation
Unbounded string copies

- The standard `strcpy()` and `strcat()` functions perform unbounded copy operations.

```c
void main(void) {
    char Password[80];
    puts("Enter 8 character password:");
    gets(Password);
    ...
}

int main(int argc, char *argv[]) {
    char name[2048];
    strcpy(name, argv[1]);
    strcat(name, "=");
    strcat(name, argv[2]);
    ...
}
```
Off-by-one errors

- 1. Copy 11 bytes, including a one-byte terminating null character.
- 2. `strlen()` does not account for the null byte.
- 3. First position in a C array is indexed by 0.

```c
int main(int argc, char *argv[]) {
    char source[10];
    strcpy(source, "0123456789");
    char *dest = (char *)malloc(strlen(source));
    for(int i=1; i<=11; i++) {
        dest[i] = source[i];
    }
    dest[11] = '\0';
    printf("dest = %s", dest);
}
```
Null termination errors

- A common problem with C-style strings is a failure to properly null terminate.

```c
int main(int argc, char *argv[]) {
    char a[16];
    char b[16];
    char c[32];

    strcpy(a, "0123456789abcdef");
    strcpy(b, "0123456789abcdef");
    strcpy(c, a); // This call to strcpy is incorrect.
    strcat(c, b);
    printf("a = %s\n", a);
    return 0;
}
```
String truncation occurs when a destination character array is not large enough to hold the contents of a string.

```cpp
#include <iostream.h>
int main() {
    char buf[12];
    cin.width(12);
    cin >> buf;
    cout << "echo: " << buf << endl;
}
```
String errors without functions

- Highly susceptible to error functions: `strcpy()`, `strcat()`, `gets()`, `streadd()`, `strncpy()`, `strtrns()`.

```c
int main(int argc, char *argv[]) {
    int i = 0;
    char buff[128];
    char *arg1 = argv[1];

    while(arg1[i] != '\0') {
        buff[i] = arg1[i];
        i++;
    }

    buff[i] = '\0';
    printf("buff = %s\n", buff);
}
```
Mitigation strategies

- Input validation

```c
int myfunc(const char *arg) {
    char buff[100];
    if(strlen(arg) >= sizeof(buff)) {
        abort();
    }
}
```
Mitigation strategies

- Use fgets() and gets_s() instead of gets()
  - Never use gets().
  - fgets(buff, BUFFSIZE, stdin)
  - gets_s(buff, BUFFSIZE)

- Use memcpy_s() and memmove_s() instead of memcpy() and memmove()
  - Add an additional argument that specifies the maximum size of the destination.
Mitigation strategies

- Use `strcpy_s()` and `strcat_s()` instead of `strcpy()` and `strcat()`
  - `strcpy_s()` `strcat_s()` : only succeeds when the source string can be fully copied to the destination without overflowing the destination buffer.
  - `strncpy()` `strncat()`: 
    - `strncpy(dest, source ,dest_size – 1);`
    - `strncat(dest, source, dest_size-strlen(dest)-1);`
  - `strncpy_s()` `strncat_s()`
  - `strlcpy()` `strlcat()`
    - `size_t strlcpy(char *dst, const char *src, size_t size);`
    - `size_t strlcat(char *dst, const char *src, size_t size);`
Detection and recovery

- Compiler generated runtime checks
- Nonexecutable stacks
- Stackgap
  - Introducing a randomly sized gap of space upon allocation of stack memory makes it more difficult for an attacker to locate a return value on stack.
- Runtime bound checkers
- Canaries
  - The canary is initialized immediately after the return address is saved and checked immediately before the return address is accessed.
Detection and recovery

- Stack smashing protector
  - Check guard value while return.
  - (A) no array
  - (B) array
  - (C) no array

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>Local variables(C)</td>
</tr>
<tr>
<td>Arrays (B)</td>
</tr>
<tr>
<td>guard</td>
</tr>
<tr>
<td>Previous frame pointer</td>
</tr>
<tr>
<td>Return address</td>
</tr>
<tr>
<td>Arguments (A)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Stack</td>
</tr>
</tbody>
</table>
Pointer subterfuge

0x00ffdeff
0x00ffde30

Array[10]
0x00ffde38

Shellcode[200]
0x00ffdeff
Data location

- UNIX executables contain both a data and a BSS segment.
- Data segment contains all initialized global variables and constants.
- BSS segment contains all uninitialized global variables.

Function pointers

Data pointers
Pointer subterfuge

- Modifying the instruction pointer
  - The instruction pointer register (eip) contains the offset in the current code segment for the next instruction to be executed.

- Global offset table
  - ELF: the default binary format on Linux, Solaris 2.x and SVR4 is called the executable and linking format (ELF).
  - The process space of any ELF binary includes a section called the global offset table (GOT). The GOT holds the absolute addresses, making them available without compromising the position independence of, and the ability to share, the program text.
**Pointer subterfuge**

- **Global offset table**

```
%objdump --dynamic-reloc test-prog
Format: file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
08049c0 R_386_GLOB_DAT __gmon_start__
08049a8 R_386_JUMP_SLOT __libc_start_main
08049ac R_386_JUMP_SLOT strcat
08049b0 R_386_JUMP_SLOT printf
08049b4 R_386_JUMP_SLOT exit
```
The .dtors section

```c
#include <stdio.h>
#include <stdlib.h>
static void create(void)
  __attribute__((constructor));
static void destroy(void)
  __attribute__((destructor));

int main(int argc, char *argv[]) {
  printf(......
  ...
}
void create(void) { ..... }
void destroy(void) { ... }
```
Pointer subterfuge

- The `.dtors` section
  - `0xffffffff {function-address} 0x00000000`

```bash
%objdump -s -j .dtors dtors

dtors: file format elf32-i386

Contents of section .dtors:
804959c ffffffff b8830480 00000000
```
The atexit() and on_exit() functions

- The atexit() function registers a function to be called without arguments at normal termination.
- The atexit() function works by adding a specified function to an array of existing functions to be called on exit. When exit() is called, it invokes each function in the array LIFO order.
Pointer subterfuge

- The longjmp() function
- Exception handling
Mitigation strategies

- The best way to prevent pointer subterfuge is to eliminate the vulnerabilities that allow memory to be improperly overwritten.
Secure Coding In C and C++
- Robert C. Seacord  Addison-Wesley
Google (www.google.com)
Any question?