Pointer Subterfuge

Secure Coding in C and C++, 2nd Edition

2022년 1학기

한양대학교 컴퓨터소프트웨어학부
임을규

Introduction

- Pointer subterfuge == modifying pointer values
- A pointer is a variable that contains the address of a function, array element, or other data structure
- Pointers to objects vs. pointers to functions
  - Function pointers can be overwritten to transfer control to attacker-supplied shellcode
  - Data pointers can also be modified to run arbitrary code.
    - attackers can control the address to modify other memory locations.
- C++ also defines pointer to member type
- First examine relationship data declaration/storage

Data Locations

- Overwriting a pointer with a buffer overflow:
  - Limited by upper bound
  - Limited by lower bound
  - Limited by Hi
  - Limited by Lo
  - Limited by special marker (usually null)
- For a buffer overflow to overwrite a function / data pointer, the buffer must be
  - allocated in the same segment as the target function/data pointer.
  - at a lower memory address than the target function/data pointer.
    - Pointer must be in direction of overflow
  - susceptible to a buffer overflow exploit
    - Buffer not adequately bounded

Data declaration and process memory organization

- UNIX executables contain both a data and a BSS segment.
  - The data segment contains all initialized global variables and constants.
  - The Block Started by Symbols (BSS) segment contains all uninitialized global variables.
  - example 3.1

```
static int GLOBAL_INIT = 1; /* data segment, global */
static int global_uninit; /* BSS segment, global */

int main(int argc, char **argv) { /* stack, local */
  int local_init = 1; /* stack, local */
  int local_uninit; /* stack, local */
  static int local_static_init = 1; /* data seg, local */
  static int local_static_uninit; /* BSS segment, local */
  /* storage for buff_ptr is stack, local */
  /* allocated memory is heap, local */
  int *buff_ptr = (int *)malloc(32);
}
```

- Initialized global variables are separated from uninitialized variables.

Function pointers – Example Program 1


```
1. void good_function(const char *str) {...}
2. void main(int argc, char **argv) {
3.     static char buff[BUFFSIZE];
4.     static void (*funcPtr)(const char *str);
5.     funcPtr = &good_function;
6.     strncpy(buff, argv[1], strlen(argv[1]));
7.     (void)(*funcPtr)(argv[2]);
8. }
```

The static character array buff

funcPtr declared are both uninitialized and stored in the BSS segment

Function Pointers - Example Program 2


```
1. void good_function(const char *str) {...}
2. void main(int argc, char **argv) {
3.     static char buff[BUFFSIZE];
4.     static void (*funcPtr)(const char *str);
5.     funcPtr = &good_function;
6.     strncpy(buff, argv[1], strlen(argv[1]));
7.     (void)(*funcPtr)(argv[2]);
8. }
```

A buffer overflow occurs when the length of argv[1] exceeds BUFFSIZE

When the program invokes the function identified by funcPtr, the shellcode is invoked instead of good_function().

Data Pointers

- Used in C and C++ to refer to
  - dynamically allocated structures
  - call-by-reference function arguments
  - arrays
  - other data structures
- Arbitrary Memory Write occurs when an Attacker can control an address to modify other memory locations

Example – Data Pointers

- By overflowing the buffer, an attacker can overwrite ptr and val.

```
1. void foo(void * arg, size_t len) {
2.     char buff[100];
3.     long val = ...;
4.     long *ptr = ...;
5.     memcpy(buff, arg, len); //unbounded memory copy
6.     *ptr = val;
7.     ...
8.     return;
9. }
```

When *ptr = val is evaluated (line 6), an arbitrary memory write is performed

Modifying the Instruction Pointer

- For an attacker to succeed an exploit needs to modify the value of the instruction pointer to reference the shellcode.

```
void good_function(const char *str) {
    printf("%s", str);
}
int _t main(int argc, _TCHAR* argv[]) {
    static void (*funcPtr)(const char *str); // Function pointer declaration
    funcPtr = &good_function;
    (void)(*funcPtr)("hi ");
    good_function("there!\n");
    return 0;
}
```

Disassembly Example


```
(void)(*funcPtr)("hi ");
00424178 mov esi, esp
0042417A push offset string "hi" (46802Ch)
0042417F call dword ptr [funcPtr (478400h)]
00424185 add  esp, 4
00424188 cmp  esi, esp

good_function("there!\n");
0042418F push offset string "there!\n" (468020h)
00424194 call good_function (422479h)
00424199 add  esp, 4
```

This address can also be found in the dword ptr [funcPtr]

First function call invocation takes place at 0x0042417F. The machine code at this address is ff 15 00 84 47 00

The actual address of good_function() stored at this address is 0x00422479

Disassembly Example

```
(void)(*funcPtr)("hi ");
00424178 mov esi, esp
0042417A push offset string "hi" (46802Ch)
0042417F call dword ptr [funcPtr (478400h)]
00424185 add  esp, 4
00424188 cmp  esi, esp
```

```
good_function("there!\n");
0042418F push offset string "there!\n" (468020h)
00424194 call good_function (422479h)
00424199 add  esp, 4
```

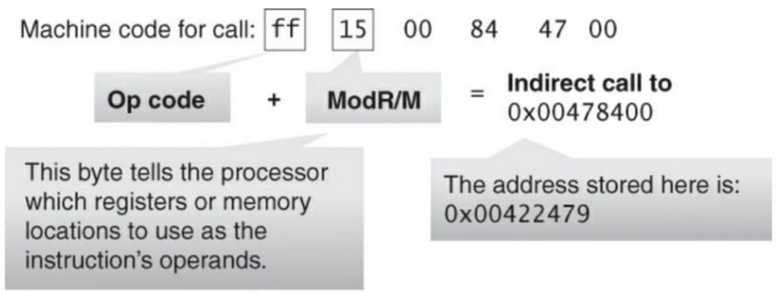

Figure 3.1. x86-32 call instruction

The second, static call to good_function() takes place at 0x00424194. The machine code at this location is e8 e0 e2 ff ff

disassembly analysis

call Goodfunction(..)

- indicates a near call with a displacement relative to the next instruction.
- The displacement is a negative number, which means that good_function() appears at a lower address
- The invocations of good_function() provide examples of call instructions that can and cannot be attacked

disassembly analysis

- The static invocation uses an immediate value as relative displacement,
  - this displacement cannot be overwritten because it is in the code segment.
- The invocation through the function pointer uses an **indirect reference**,
  - the address in the referenced location can be overwritten.
  - These indirect function references can be exploited to transfer control to arbitrary code.

Global Offset Table

- Windows and Linux use a similar mechanism for linking and transferring control to library functions
- Windows solution is safe
- Linux solution is exploitable
  - Default binary format on Linux is called **Executable and Linking Format (ELF)**
  - Developed by Unix System Labs as part of the application binary interface
  - Includes a "**Global Offset Table**" (GOT)

How it works

- Holds absolute addresses of library functions
  - program text is still position independent
  - program text can still be shared
    - essential for the dynamic linking process to work
- Initially entry to Run-Time Linker
- Every library function used by a program has an entry in the GOT that contains the address of the actual function.
  - Before the program uses a function for the first time, the entry contains the address of the runtime linker (RTL).
  - If the function is called by the program, control is passed to the RTL and the function's real address is resolved and inserted into the GOT.
  - Subsequent calls invoke the function directly through the GOT entry without involving the RTL

How it works

- Address of GOT is fixed
- Address of GOT entry is fixed in the ELF executable
- The GOT entry is at the same address for any executable process image
  - Obtainable through `objdump -dynamic-reloc xx` command (undocumented!!)
  - [참고]
    - `objdump -h xx` → section 정보를 확인할 수 있음
    - `objdump -d xx` → disassembled code를 확인할 수 있음
- An attacker can overwrite a GOT entry for a function with the address of shellcode using an arbitrary memory write.
  - Control is transferred to the shellcode when the program subsequently invokes the function corresponding to the compromised GOT entry.

Example

- % objdump --dynamic-reloc test-prog
- format: file format elf32-i386
- **DYNAMIC RELOCATION RECORDS**
- **OFFSET TYPE VALUE**
- `08049bc0 R_386_GLOB_DAT __gmon_start__`
- `08049ba8 R_386_JUMP_SLOT __libc_start_main`
- `08049bac R_386_JUMP_SLOT strcat`
- `08049bb0 R_386_JUMP_SLOT printf`
- **`08049bb4 R_386_JUMP_SLOT exit`**
- `08049bb8 R_386_JUMP_SLOT sprintf`
- `08049bbc R_386_JUMP_SLOT strcpy`

The offsets specified for each R_386_JUMP_SLOT relocation record contain the address of the specified function (or the RTL linking function)

Global Offset Table (GOT)

- Windows portable executable (PE) file format is similar to ELF:
  - Array of data structures for each imported DLL
    - Name → array of function pointers (Import Address Table, IAT)
    - Once module is loaded (at load time), IAT entries are **write protected**

The .dtors Section

- Another function pointer attack is to overwrite function pointers in the `.dtors` section for executables generated by GCC
- GNU C allows a programmer to declare attributes about functions by specifying the `__attribute__` keyword followed by an attribute specification inside double parentheses
- Attribute specifications include **constructor** and **destructor**.
- The constructor attribute specifies that the function is called before `main()`
- The destructor attribute specifies that the function is called after `main()` has completed or `exit()` has been called.

Example


```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("create: %p.\n", create);
    printf("destroy: %p.\n", destroy);
  exit(EXIT_SUCCESS);
}
void create(void) {
    printf("create called.\n");
}
void destroy(void) {
    printf("destroy called.\n");
}
```

create called
create: 0x80483a0

destroy: 0x80483b8
destroy called

Example 3.8. Output of Sample Program

```
% ./dtors
create called.
create: 0x80483a0.
destroy: 0x80483b8.
destroy called.
```

The .dtors section

- Constructors and destructors are stored in the .ctors and .dtors sections in the generated ELF executable image.
- Both sections have the following layout:
  - 0xffffffff {function-address} 0x00000000
- The .ctors and .dtors sections are mapped into the process address space and are writable by default.
- Constructors have not been used in exploits because they are called before the main program.
- The focus is on destructors and the .dtors section.
- The contents of the .dtors section in the executable image can be examined with the objdump command
  - objdump -s -j .dtors <fname>

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

dtors:     file format elf32-i386

Contents of section .dtors:
804959c ffffffff b8830408 00000000
```

The .dtors section

- An attacker can transfer control to arbitrary code by overwriting the address of the function pointer in the `.dtors` section.
- If the target binary is readable by an attacker, an attacker can find the exact position to overwrite by analyzing the ELF image.
- The `.dtors` section is present even if no destructor is specified.
  - The `.dtors` section consists of the head and tail tag with no function addresses between.
  - It is still possible to transfer control by overwriting the tail tag 0x00000000 with the address of the shellcode.

The .dtors section

- For an attacker, the `.dtors` section has advantages over other targets:
  - `.dtors` is always present and mapped into memory.
  - It is difficult to find a location to inject the shellcode onto so that it remains in memory after main() has exited.
  - The `.dtors` target only exists in programs that have been compiled and linked with GCC.

Examples

- good function

```
#include <stdio.h>

void good_function(const char *str) {
	printf("%s", str);
}
int main(int argc, char *argv[]) {
	void (*funcPtr)(const char *);
	funcPtr = &good_function;
	(*funcPtr)("Hi, ");
	good_function("there!\n");

	return 0;
}
```

- .dtors

```
#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("create: %p.\n", create);
	printf("destroy: %p.\n", destroy);
	exit(EXIT_SUCCESS);
}
void create(void) {
	printf("create called.\n");
}
void destroy(void) {
	printf("destroy called.\n");
}
```

Virtual Pointers

- A virtual function is a function member of a class, declared using the virtual keyword.
- Functions may be overridden by a function of the same name in a derived class.
- A pointer to a derived class object may be assigned to a base class pointer, and the function called through the pointer.
- Without virtual functions, the base class function is called because it is associated with the static type of the pointer.
- When using virtual functions, the derived class function is called because it is associated with the dynamic type of the object

Example


```
class a {
  public:
    void f(void) {
      cout << "base f" << endl;
    };
    virtual void g(void) {
      cout << "base g" << endl;
    };
};

class b: public a {
  public:
    void f(void) {
      cout << "derived f" << endl;
    };
    void g(void) {
      cout << "derived g" << endl;
    };
};

int _tmain(int argc, _TCHAR* argv[]) {
  a *my_b = new b();
  my_b->f();
  my_b->g();
  return; }
```

`Class a` is defined as the base class and contains a regular function `f()` and a virtual function `g()`.

`Class b` is derived from `class a` and overrides both functions

A pointer `my_b` to the base class is declared in main() but assigned to an object of the derived `class b`

Example


```
int _tmain(int argc, _TCHAR* argv[]) {
  a *my_b = new b();
  my_b->f();
  my_b->g();
  return; }
```

A pointer `my_b` to the base class is declared in `main()` but assigned to an object of the derived `class b`

When the non-virtual function `my_b->f()` is called on the function `f()` associated with `a` (the base class) is called.

When the virtual function `my_b->g()` is called on the function `g()` associated with `b` (the derived class) is called

Virtual Pointers

- Most C++ compilers implement virtual functions using a **virtual function table (VTBL)**.
- The VTBL is *an array of function pointers* that is used at runtime for dispatching virtual function calls.
- Each individual object points to the VTBL via a virtual pointer (VPTR) in the object’s header.
- The VTBL contains pointers to each implementation of a virtual function

Virtual Pointers

- It is possible to overwrite function pointers in the VTBL or to change the VPTR to point to another arbitrary VTBL.
- This can be accomplished by an arbitrary memory write or by a buffer overflow directly into an object.
- The buffer overwrites the VPTR and VTBL of the object and allows the attacker to cause function pointers to execute arbitrary code

VTBL Runtime Representation

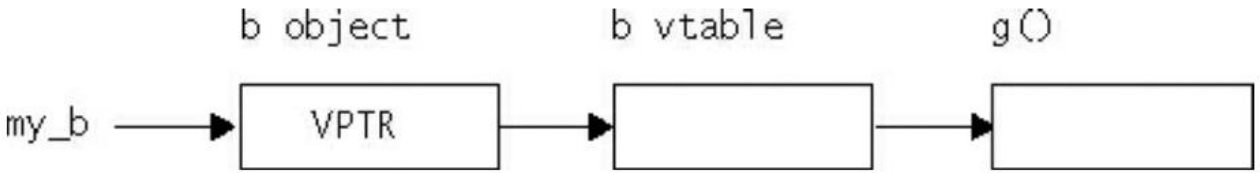

Figure 3.2. VTBL runtime representation

atexit() and on_exit()

- The `atexit()` function is a general utility function defined in C99.
- The `atexit()` function registers a function to be called without arguments at normal program termination.
- C99 requires that the implementation support the registration of at least 32 functions.
- The `on_exit()` function from SunOS performs a similar function.
- This function is also present in libc4, libc5, and glibc

the atexit() and on_exit() functions

- 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 in last in, first out (LIFO) order.
- Because both `atexit()` and `exit()` need to access this array, it is allocated as a global symbol (`__atexit` on *bsd and `__exit_funcs` on Linux)

Example

```
1. char *glob;
2. void test(void) {
3.     printf("%s", glob);
4. }
5. void main(void) {
6.     atexit(test);
7.     glob = "Exiting.\n";
8. }
```

actual function addresses

```
(gdb) b main
Breakpoint 1 at 0x80483f6: file atexit.c, line 6.
(gdb) r
Starting program: /home/rcs/book/dtors/atexit

Breakpoint 1, main (argc=1, argv=0xbfffe744) at atexit.c:6
6 atexit(test);
(gdb) next
7 glob = "Exiting.\n";
(gdb) x/12x __exit_funcs
0x42130ee0 <init>:    0x00000000 0x00000003 0x00000004 0x4000c660
0x42130ef0 <init+16>: 0x00000000 0x00000000 0x00000004 0x0804844c
0x42130f00 <init+32>: 0x00000000 0x00000000 0x00000004 0x080483c8
(gdb) x/4x 0x4000c660
0x4000c660 <_dl_fini>: 0x57e58955 0x5ce85356 0x81000054 0x0091c1c3
(gdb) x/3x 0x0804844c
0x804844c <__libc_csu_fini>: 0x53e58955 0x9510b850 x102d0804
(gdb) x/8x 0x080483c8
0x80483c8 <test>: 0x83e58955 0xec8308ec 0x2035ff08 0x68080496
```

debugging using gdb

- Three functions have been registered: `_dl_fini()`, `__libc_csu_fini()`, `test()`.
- It is possible to transfer control to arbitrary code with an arbitrary memory write or a buffer overflow directly into the `__exit_funcs` structure.
- The `_dl_fini()` and `__libc_csu_fini()` functions are present even when the vulnerable program does not explicitly call the `atexit()` function

longjmp()

- C99 defines the `setjmp()` macro, `longjmp()` function, and `jmp_buf` type, which can be used to bypass the normal function call and return discipline.
- The `setjmp()` macro saves its calling environment for later use by the `longjmp()` function.
- The `longjmp()` function restores the environment saved by the most recent invocation of the `setjmp()` macro

Example


```
1. typedef int __jmp_buf[6];
2. #define JB_BX 0
3. #define JB_SI 1
4. #define JB_DI 2
5. #define JB_BP 3
6. #define JB_SP 4
7. #define JB_PC 5
8. #define JB_SIZE 24
9. typedef struct __jmp_buf_tag {
10.     __jmp_buf __jmpbuf;
11.     int __mask_was_saved;
12.     __sigset_t __saved_mask;
13. } jmp_buf[1]
```

- The `jmp_buf` structure (lines 9-13) contains three fields.
- The calling environment is stored in `__jmpbuf` (declared on line 1).
- The `__jmp_buf` type is an integer array containing six elements.
- The `#define` statements indicate which values are stored in each array element.
- The base pointer (BP) is stored in `__jmp_buf[3]`,
- The program counter (PC) is stored in `__jmp_buf[5]`

Example – Assembly instructions generated for longjmp() on Linux

longjmp(env, i)

1. movl i, %eax /* return i */
2. movl env.__jmpbuf[JB_BP], %ebp
3. movl env.__jmpbuf[JB_SP], %esp
4. jmp (env.__jmpbuf[JB_PC])

The movl instruction on line 2 restores the BP

The movl instruction on line 3 restores the stack pointer (SP)

Line 4 transfers control to the stored PC

the longjump() function

- The `longjmp()` function can be exploited by overwriting the value of the PC in the `jmp_buf` buffer with the start of the shellcode.
- This can be accomplished with an arbitrary memory write or by a buffer overflow directly into a `jmp_buf` structure

Example

- longjump.c

```
#include <setjmp.h>
jmp_buf buf;
void g(int n);
void h(int n);
int n = 6;
void f(void) {
    setjmp(buf);
    g(n);
}
void g(int n) {
    h(n);
}
void h(int n){
    if (n-- > 0)
        longjmp(buf, 2);
}
```

- atexit.c

```
#include <stdio.h>
#include <stdlib.h>
char *glob;
void test(void) {
        printf("%s", glob);
}
int main(void) {
        atexit(test);
        glob = "Exiting.\n";
}
```

Exception Handling

- Windows has three types:
  - Vectored exception handling
  - Structured exception handling (try/catch)
  - System defaults
- Unix has three:
  - Vectored exception handling
  - Structured exception handling (try/catch)
  - System defaults (see `man signal`, `man sigprocmask`)

Structured Exception Handling

- try … catch blocks

```
try {
  // Do stuff here
}
catch(...){
  // Handle exception here
}
__finally {
  // Handle cleanup here
}
```

- Stack frame initialization with exception handler

```
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF8h
push    0FFFFFFFFh
push    ptr [Exception_Handler]
mov     eax, dword ptr fs:[00000000h]
push    eax
mov     dword ptr fs:[0], esp
```

Stack frame

- Stack frame initialization with exception handler

Table 3.1. Stack Frame with Exception Handler

Stack Offset	Description	Value
−0x10	Handler	[Exception_Handler]
−0x0C	Previous handler	fs:[0] at function start
−8	Guard	−1
−4	Saved	ebp ebp
0	Return address	Return address

Mitigation Strategies

- The best way to prevent pointer subterfuge is to eliminate the vulnerabilities that allow memory to be improperly overwritten.
  - Pointer subterfuge can occur as a result of
    - Overwriting data pointers
    - Common errors managing dynamic memory
    - Format string vulnerabilities
- Eliminating these sources of vulnerabilities is the best way to eliminate pointer subterfuge.
- Eliminate the vulnerabilities:
  - Stack canaries
  - W ^ X
  - Encode/decode function pointers

W ^ X

- One way to limit the exposure from some of these targets is to reduce the privileges of the vulnerable processes.
  - The policy called “W xor X” or “W^X” states that a memory segment may be writable or executable, but not both.
  - It is not clear how this policy can be effectively enforced to prevent overwriting targets such as atexit() that need to be both writable at runtime and executable

Encode/decode function pointers

- Encode function pointers

```
#include <stdlib.h>
void (*)() encode_pointer(void(*pf)());
```

Description

The `encode_pointer()` function performs a transformation on the `pf` argument, such that the `decode_pointer()` function reverses that transformation.

Returns

The result of the transformation.

- decode function pointers

```
#include <stdlib.h>
void (*)() decode_pointer(void(*epf)());
```

Description

The `decode_pointer()` function reverses the transformation performed by the `encode_pointer()` function.

Returns

The result of the inverse transformation.

- make buffer overflows more difficult to exploit


Q&A