Pointer Subterfuge

- A *pointer* is a variable that contains the address of a function, array element, or other data structure.
- 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.

Data Locations-1

- 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.
  - susceptible to a buffer overflow exploit.

Data Locations-1

- 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.
- Initialized global variables are separated from uninitialized variables.

Data Declarations and Process Organization

```
1. static int GLOBAL_INIT = 1;          /* data segment, global */
2. static int global_uninit;            /* BSS segment, global */
3.
4. void main(int argc, char **argv) {   /* stack, local */
5.   int local_init = 1;                /* stack, local */
6.   int local_uninit;                  /* stack, local */
7.   static int local_static_init = 1; /* data seg, local */
8.   static int local_static_uninit;   /* BSS segment, local*/
     /* storage for buff_ptr is stack, local */
     /* allocated memory is heap, local */
9.   }
```

Function Pointers

```
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 Pointer—Example 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

Data Pointers--Example

```
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. }
```

- By overflowing the buffer, an attacker can overwrite ptr and val.
- When *ptr = val is evaluated (line 6), an arbitrary memory write is performed.

Modifying an Instruction

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

```
1. void good_function(const char *str) {
2.   printf("%s", str);
3. }
4. int _tmain(int argc, _TCHAR* argv[]) {
5.   static void (*funcPtr)(const char *str);
     // Function pointer declaration
6.   funcPtr = &good_function;
7.   (void)(*funcPtr)("hi ");
8.   good_function("there!\n");
9.   return 0;
10. }
```

Modifying an Instruction

The instruction pointer is a special register in a CPU that keeps track of the address of the next instruction to be executed. In x86 architecture, it is referred to as the Instruction Pointer (EIP/RIP),while in ARM architecture, it is known as the Program Counter (PC).

Modifying an Instruction

In standard C and C++, directly modifying the instruction pointer (IP)—such as by accessing and changing the EIP or RIP register—is not supported. The language itself does not provide any mechanisms to directly manipulate CPU registers like the instruction pointer. C and C++ are high-level languages, so they abstract away most of the low-level details of CPU registers.

indirect ways to influence the program's control flow do exist:

Function Pointers: By changing which function pointer you call, you can influence where execution jumps, effectively altering the flow of the program.

setjmp and longjmp: These standard library functions allow you to "jump" back to a saved point in the program, effectively moving to a different instruction location without directly modifying the IP.

Inline Assembly (with GCC or MSVC): Some compilers support inline assembly, which allows you to write assembly instructions directly within C/C++ code.

goto Statement: While not a modification of the IP, goto can redirect the flow of control, which might have a similar effect on program execution.

Modifying an Instruction

Example: Assembly Language

```
section .text
global _start
_start:
mov eax, 1 ; syscall: sys_exit
jmp end_label ; jump to end_label

; This code will be skipped
mov ebx, 0 ; exit code 0

end_label:
int 0x80 ; invoke syscall
```

In this example, the jmp instruction alters the instruction pointer to skip the line that sets the exit code, going directly to end_label.

Function pointer Disassembly Program

```
(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

Function pointer Disassembly Program-

```
(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_fu
00424199 add  esp, 4
```

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

Disassembly

Disassembly is the process of translating machine code (binary code executed by the CPU) back into human-readable assembly language. This reverse engineering process helps developers and engineers understand what a compiled program does at a low level, especially when source code is not available or when debugging complex issues.

Disassembly Program

- Helps identify bugs, especially in low-level programming (like operating systems or drivers).
- Useful in cybersecurity, forensics, and understanding malware behaviour.
- Assembly reveals what code a compiler produces, allowing developers to optimize code for performance.

Tools for Disassembly:

Objdump: A common tool on UNIX-like systems for disassembling binaries.
GDB: The GNU Debugger has built-in disassembly functionality.
IDA Pro and Ghidra: Advanced tools for disassembly and interactive analysis, popular in reverse engineering.

Function Pointer Disassembly Analysis-1

- 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

Function Pointer Disassembly Program –Analysis 2

- 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 -1

- Windows and Linux use a similar mechanism for linking and transferring control to library functions.
  - Linux solution is exploitable
  - Windows version is not
- The default binary format on Linux, Solaris 2.x, and SVR4 is called the executable and linking format (ELF).
- ELF was originally developed and published by UNIX System Laboratories (USL) as part of the application binary interface (ABI).
- The ELF standard was adopted by the Tool Interface Standards committee (TIS) as a portable object file format for a variety of IA-32 operating systems.

Global Offset Table -1

- The Global Offset Table (GOT) is a crucial data structure used in dynamic linking for programs, especially in systems that use the Executable and Linkable Format (ELF), like many Unix-based systems. It facilitates the management of dynamically linked libraries (shared libraries) and allows functions and variables in shared libraries to be accessed at runtime.

- The Global Offset Table is an array of pointers (addresses) that holds the addresses of global variables and functions that are defined in shared libraries. It is created during the linking process of a program.

- The GOT allows a program to access variables and functions in shared libraries without knowing their actual memory addresses at compile time. This is essential for dynamic linking, where the actual addresses may vary between program runs.

Global Offset Table -1

How the GOT Works

1. **Initialization**:

• When a program using dynamic linking starts, the dynamic linker/loader (like ld.so in Linux) loads the required shared libraries into memory.

• The GOT is initialized with the addresses of the functions and variables that the program will use from these libraries.

2. **Accessing Functions and Variables** :

• When a function or variable in a shared library is called for the first time, the program accesses it through the GOT.

• The GOT entry is updated to point to the actual address of the function or variable in memory.

3. **Relocation**:

• During the linking process, the GOT is populated with placeholder addresses. The dynamic linker resolves these placeholders with the actual addresses when the program is loaded.

Global Offset Table -2

- The process space of any ELF binary includes a section called the global offset table (GOT).
  - The GOT holds the absolute addresses,
  - Provides ability to share, the program text.
    - essential for the dynamic linking process to work.
- 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

Global Offset Table -3

- The address of a GOT entry is fixed in the ELF executable.
- The GOT entry is at the same address for any executable process image.
- The location of the GOT entry for a function can be found using the objdump
- 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.

Global Offset Table 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_SI
```

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

.dtors Section

- The .dtors section, short for "destructors," is a part of the binary structure of C and C++ programs that utilize dynamic linking. It is primarily used for managing the cleanup of objects when a shared library is unloaded or when a program terminates.

- The .dtors section is a part of the Executable and Linkable Format (ELF) used on Unix-like operating systems. It contains pointers to destructor functions for objects that need to be cleaned up when a program or library is exiting.

- The main purpose of the .dtors section is to ensure that destructors for global and static objects are called in the correct order when a program or shared library is unloaded. This is particularly important in C++ where resource management is often handled through destructors.

.dtors Section

```
#include <iostream>
class Resource {
public:
Resource() {
std::cout << "Resource acquired." << std::endl;
}
~Resource() {
std::cout << "Resource released." << std::endl;
}
};
Resource globalResource; // Global object
int main() {
std::cout << "In main." << std::endl;
return 0;
}
```

In this example:

• When the program starts, globalResource is constructed, and the message "Resource acquired." is printed.

• Upon termination of the program, the destructor for globalResource is called, printing "Resource released." This destructor would be registered in the .dtors section.

.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.

.dtors Section--Example

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

3. static void create(void)
        __attribute__ ((constructor));
4. static void destroy(void)
        __attribute__ ((destructor));

5. int main(int argc, char *argv[]) {
6.   printf("create: %p.\n", create);
7.   printf("destroy: %p.\n", destroy);
8.   exit(EXIT_SUCCESS);
9. }

10. void create(void) {
11.   printf("create called.\n");
12. }

13. void destroy(void) {
14.   printf("destroy called.\n");
15. }
```

create called.
create: 0x80483a0.

destroy: 0x80483b8.
destroy called.

.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

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

Virtual Pointers

```
1. class a {
2.   public:
3.     void f(void) {
4.       cout << "base f" << endl;
5.     };

6.     virtual void g(void) {
7.       cout << "base g" << endl;
8.     };
9. };

10. class b: public a {
11.   public:
12.     void f(void) {
13.       cout << "derived f" << endl;
14.     };

15.     void g(void) {
16.       cout << "derived g" << endl;
17.     };
18. };

19. int _tmain(int argc, _TCHAR* argv[])
20.   a *my_b = new b();
21.   my_b->f();
22.   my_b->g();
23.   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 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.

Virtual Pointers

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

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

Virtual functions enable polymorphism, where a base class reference or pointer can invoke derived class implementations of functions. This allows the program to decide at runtime which function to execute based on the actual object type.

When a virtual function is called through a base class pointer or reference, the call is resolved at runtime, not at compile time. This is called dynamic binding or late binding, meaning the exact function executed depends on the object's actual type rather than the pointer/reference type.

In C++, the keyword virtual is used in the base class to declare a function as virtual. The derived classes can then override this function, and calls to this function will be resolved at runtime.

Derived classes can provide their specific implementations of a base class's virtual functions. If a derived class doesn't override a virtual function, the base class's version is used.

Virtual Functions

```
#include <iostream>
using namespace std;

class Base {
public:
    virtual void show() {  // Virtual function
        cout << "Base class show()" << endl;
    }
};

class Derived : public Base {
public:
    void show() override {  // Overridden function
        cout << "Derived class show()" << endl;
    }
};

int main() {
    Base* ptr = new Derived();  // Pointer of base type pointing to derived object
    ptr->show();  // Calls Derived::show() due to virtual function

    delete ptr;
```

Virtual Pointers

Virtual Pointer(vptr) is an internal pointer maintained by the compiler for each object of a class that contains virtual functions.

Virtual Table (vtable): Each class with virtual functions has a virtual table. This table holds pointers to the actual implementations of the class's virtual functions. If a derived class overrides these virtual functions, the derived class’s vtable will have pointers to the overridden functions.

Virtual Pointer (vptr): Inside each object of a class with virtual functions, there is a hidden pointer (the virtual pointer or vptr) that points to the vtable of that object’s class. When a virtual function is called on an object, the vptr is used to look up the function in the vtable, ensuring the correct function is called based on the object’s actual type.

How It Enables Polymorphism: By using the vptr to access the vtable, C++ allows the program to decide at runtime which function to call, rather than at compile time. This makes polymorphism possible by allowing the derived class's function to override the base class’s function.

Virtual Pointers

```
class Base {
public:
    virtual void show() {
        std::cout << "Base show" << std::endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        std::cout << "Derived show" << std::endl;
    }
};

int main() {
    Base* ptr = new Derived();
    ptr->show(); // Calls Derived::show due to vptr/vtable
    delete ptr;
}
```

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.

Virtual Pointers

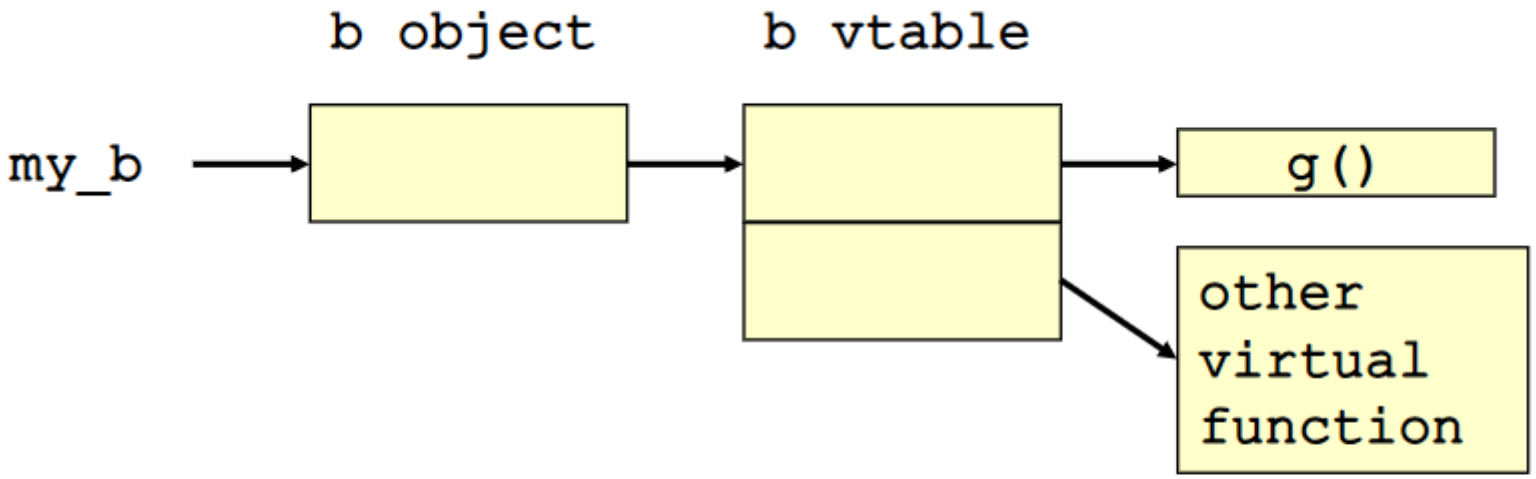
Attack Surface of vptr and vtable

Since vptr and vtable depend on pointers and memory layout, attackers can exploit vulnerabilities in these structures, especially if they gain the ability to corrupt memory near the vptr or overwrite the vtable. Here are a few common attack vectors:

Buffer Overflow Attacks
A buffer overflow happens when a program writes more data to a buffer (a fixed-length block of memory) than it can hold, which can overwrite adjacent memory, including the vptr.

If an attacker can control the memory around an object's vptr, they may overwrite it with a pointer to a malicious vtable.
By modifying the vptr, the attacker can make the program call arbitrary functions—often malicious code—instead of the intended virtual functions.

VTBL Run Time Representation



atexit() and on_exit() functions

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

atexit() and on_exit() functions

atexit() and on_exit() are two functions in C/C++ used to register functions that should be called when a program terminates.

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

void cleanup1() {
    printf("Cleanup 1\n");
}

void cleanup2() {
    printf("Cleanup 2\n");
}

int main() {
    atexit(cleanup1);
    atexit(cleanup2);
    printf("Main function\n");
    return 0;
}
```

atexit() and on_exit() functions

atexit() and on_exit() are two functions in C/C++ used to register functions that should be called when a program terminates.

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

void cleanup(int exit_status, void
*message) {
    printf("Exit status: %d, Message: %s\n",
exit_status, (char *)message);
}

int main() {
    on_exit(cleanup, "Goodbye!");
    printf("Main function\n");
    return 0;
}
```

atexit() and on_exit() functions

```
1. char *glob;

2. void test(void) {
3.    printf("%s", glob);
4. }

5. void main(void) {
6.    atexit(test);
7.   glob = "Exiting.\n";
8. }
```

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)

Debug session of atexit() using gdb

```
(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
```

longjmp() function

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.

longjmp() function

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

jmp_buf env;  // Declaring a jump buffer to save the program state

int main() {
    // Set the jump point with setjmp; if returning directly, it returns 0
    if (setjmp(env) == 0) {
        // The first time setjmp is called, it returns 0
        printf("Normal execution: Setting the jump point.\n");

        // Simulating an error
        printf("An error occurs! Jumping back to the safe point...\n");
        longjmp(env, 1);  // Jump back to the setjmp point with a non-zero value
    } else {
        // This block runs when longjmp jumps back to setjmp
        printf("Jumped back to the safe point after an error.\n");
    }
    printf("Program continues normally.\n");
    return 0;
}
```

longjmp() function

Output:

Normal execution: Setting the jump point.
An error occurs! Jumping back to the safe point...
Jumped back to the safe point after an error.
Program continues normally.

longjmp() function

```
1. #include <setjmp.h>
2. jmp_buf buf;
3. void g(int n);
4. void h(int n);
5. int n = 6;

6. void f(void) {
7.    setjmp(buf);
8.    g(n);
9. }

10. void g(int n) {
11.    h(n);
12. }

13. void h(int n){
14.    longjmp(buf, 2);
15. }
```

longjmp() 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

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.

Mitigation Strategies in Cyber Security

- **Access Control**
- **Regular Software Updates**
- **Network Security**
- **Data Encryption**
- **Security Training**
- **Incident Response Planning**

Mitigation Strategies in Software Development

- **Code Reviews**
- **Secure Coding Practices**
- **Testing and Validation**
- **Version Control and Backup**