Pointer Subterfuge

Secure Coding in C and C++

Robert C. Seacord

CERT® Coordination Center
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Agenda

Pointer Subterfuge

Function Pointers

Data Pointers

Modifying the Instruction Pointer

Examples

Mitigation Strategies

Summary

Agenda

Pointer Subterfuge

Function Pointers

Data Pointers

Modifying the Instruction Pointer

Examples

Mitigation Strategies

Summary

Pointer Subterfuge

Pointer subterfuge is a general term for exploits that modify a pointer's value.

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.

When the program executes a call via the function pointer, the attacker's code is executed instead of the intended code.

Data pointers can also be modified to run arbitrary code.

If a data pointer is used as a target for a subsequent assignment, attackers can control the address to modify other memory locations.

Data Locations - 1

For a buffer overflow to overwrite a function or data pointer the buffer must be

- allocated in the same segment as the target function or data pointer.
- at a lower memory address than the target function or data pointer.
- susceptible to a buffer overflow exploit.

Data Locations - 2

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 memory organization

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

void 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 */
  }
```

Agenda

Pointer Subterfuge

Function Pointers

Data Pointers

Modifying the Instruction Pointer

Examples

Mitigation Strategies

Summary

Function Pointers - Example Program 1

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

The static character array buff

funcPtr declared are both uninitialized and stored in the BSS segment.

Function Pointers - Example Program - 2

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

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

Function Pointers - Example Program - 3

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

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

Agenda

Pointer Subterfuge

Function Pointers

Data Pointers

Modifying the Instruction Pointer

Examples

Mitigation Strategies

Summary

Data Pointers

Used in C and C++ to refer to

- **dynamically allocated structures**
- **call-by-reference function arguments**
- **arrays**
- **other data structures**

Can be modified by an attacker when exploiting a buffer overflow vulnerability.

Arbitrary Memory Write occurs when an Attacker can control an address to modify other memory locations

Data Pointers - Example Program

```
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. }
```

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

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

Agenda

Modifying the Instruction Pointer

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. }
```

Function Pointer Disassembly Example - 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 address is 0x00422479.

Function Pointer Disassembly Example - 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
```

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

Function Pointer Disassembly Analysis - 1

This form of the call instruction 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 analysis - 2

The static invocation uses an *immediate* value as relative displacement, and this displacement cannot be overwritten because it is in the code segment.

The invocation through the function pointer uses an *indirect* reference, and the address in the referenced location can be overwritten.

These indirect function references, as well as function calls that cannot be resolved at compile time, can be exploited to transfer control to arbitrary code.

Agenda

Pointer Subterfuge

Function Pointers

Data Pointers

Modifying the Instruction Pointer

Examples
- Global Offset Table
- The `.dtors` Section
- Virtual Pointers
- The `atexit()` and `on_exit()` Functions
- The `longjmp()` Function
- Exception Handling

Mitigation Strategies

Summary

Agenda

Pointer Subterfuge
Function Pointers
Data Pointers
Modifying the Instruction Pointer
Mitigation Strategies
Summary

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 - 2

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.

This table is 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 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 - 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.

Agenda

Pointer Subterfuge

Function Pointers

Data Pointers

Modifying the Instruction Pointer

Examples
- Global Offset Table
- The `.dtors` Section
- Virtual Pointers
- The `atexit()` and `on_exit()` Functions
- The `longjmp()` Function
- Exception Handling

Mitigation Strategies

Summary

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.

The .dtors Section - Example Program

```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");
}
```

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

```
destroy: 0x80483b8.
destroy called.
```

The .dtors Section - 1

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

The .dtors Section - 2

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.

If the shellcode returns, the process will call subsequent addresses until a tail tag is encountered or a fault occurs.

The .dtors Section - 3

For an attacker, overwriting the .dtors section has advantages over other targets:

- .dtors is always present and mapped into memory.
- The .dtors target only exists in programs that have been compiled and linked with GCC.
- It is difficult to find a location to inject the shellcode onto so that it remains in memory after main() has exited.

Agenda

Virtual Pointers - 1

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 - Example Program- 1

```
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 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 - Example Program- 1

```
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 - 2

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.

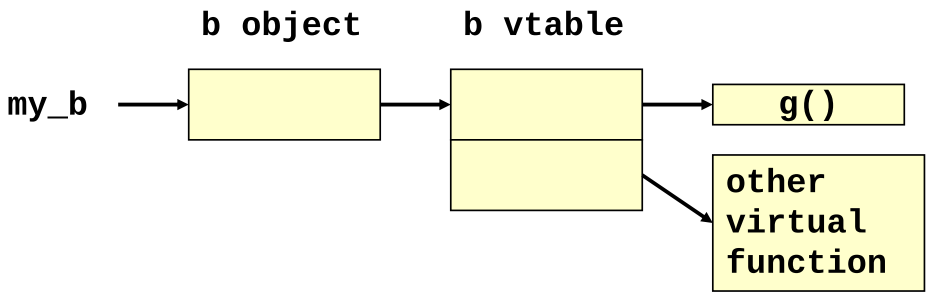
Virtual Pointers - 3

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



Agenda

Pointer Subterfuge

Function Pointers

Data Pointers

Modifying the Instruction Pointer

Examples
- **Global Offset Table**
- **The `.dtors` Section**
- **Virtual Pointers**
- **The `atexit()` and `on_exit()` Functions**
- **The `longjmp()` Function**
- **Exception Handling**

Mitigation Strategies

Summary

The atexit() and on_exit() Functions - 1

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() – Example Program

```
char *glob;

void test(void) {
  printf("%s", glob);
}

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

The atexit() and on_exit() Functions - 2

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 program using gdb - 1

```
(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
```

Debug session of atexit program using gdb - 2

In the debug session, a breakpoint is set before the call to atexit() in main() and the program is run.

The call to atexit() is then executed to register the test() function.

After the test() function is registered, memory at __exit_funcs is displayed.

Each function is contained in a structure consisting of four doublewords.

Debug session of atexit program using gdb - 3

The last doubleword in each structure contains the actual address of the function.

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.

Agenda

Pointer Subterfuge

Function Pointers

Data Pointers

Modifying the Instruction Pointer

Examples
- Global Offset Table
- The `.dtors` Section
- Virtual Pointers
- The `atexit()` and `on_exit()` Functions
- The `longjmp()` Function
- Exception Handling

Mitigation Strategies

Summary

The 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.

The longjmp() Function- Example Program - 1

```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){
  longjmp(buf, 2);
}
```

The longjmp() Function Example Program- 2

```c
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]

Carnegie Mellon Software Engineering Institute — CERT Coordination Center

The longjmp() Function Example Program- 3

```
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

Carnegie Mellon Software Engineering Institute — CERT Coordination Center

The 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

Agenda

Pointer Subterfuge

Function Pointers

Data Pointers

Modifying the Instruction Pointer

Examples
- **Global Offset Table**
- **The `.dtors` Section**
- **Virtual Pointers**
- **The `atexit()` and `on_exit()` Functions**
- **The `longjmp()` Function**
- **Exception Handling**

Mitigation Strategies

Summary

Exception Handling - 1

An exception is any event that is outside the normal operations of a procedure.

Example: Dividing by zero will generate an exception.

Exception handler blocks are used to handle special cases and avoid unexpected program termination.

Exception handlers are chained and called in a defined order until one can handle the exception.

Exception Handling - 2

Microsoft Windows exception handlers

- **Structured exception handling** (SEH). These are implemented as per-function or per-thread exception handlers.
- **System default exception handling.** This is a global exception filter and handler for the entire process that is called if no previous exception handler can handle the exception.
- **Vectored exception handling (VEH).** These are called first as a method to override a structured exception handler.

Structured Exception Handling - 1

Structured exception handling (SEH) is typically implemented at the compiler level through try…catch blocks

Any exception that is raised during the try block is handled in the matching catch block.

If the catch block is unable to handle the exception, it is passed back to the prior scope block.

The __finally keyword is a Microsoft extension to the C/C++ language and is used to denote a block of code that is called to clean up anything instantiated by the try block.

The keyword is called regardless of how the try block exits.

Try...catch block

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

Structured Exception Handling - 2

Windows implements special support for per-thread exception handlers.

Compiler-generated code writes the address of a pointer to an EXCEPTION_REGISTRATION structure to the address referenced by the fs segment register.

This structure is defined in the assembly language struc definition in EXSUPP.INC in the Visual C++ runtime source

EXCEPTION_REGISTRATION struc definition

```
EXCEPTION_REGISTRATION struc
    prev                dd      ?
    handler             dd      ?
_EXCEPTION_REGISTRATION ends
```

prev is a pointer to the previous EXCEPTION_HANDLER structure in the chain

handler is a pointer to the actual exception handler function

Windows exception handler rules

The EXCEPTION_REGISTRATION structure must be located on the stack.

The prev EXCEPTION_REGISTRATION structure must be at a higher stack address.

The EXCEPTION_REGISTRATION structure must be aligned on a doubleword boundary.

If the executable image header lists SAFE SEH handler addresses, the handler address must be listed as a SAFE SEH handler.

If the executable image header does not list SAFE SEH handler addresses, any structured exception handler may be called.

Stack frame initialization

```
1. push      ebp
2. mov       ebp, esp
3. and       esp, 0FFFFFFF8h
4. push      0FFFFFFFFh
5. push      ptr [Exception_Handler]
6. mov       eax, dword ptr fs:[00000000h]
7. push      eax
8. mov       dword ptr fs:[0], esp
```

Stack frame with exception handler

The compiler initializes the stack frame in the function prolog

The compiler reserves space on the stack for local variables.

The exception handler address could be overwritten by an arbitrary value resulting from a buffer overflow in a stack variable

It is possible to replace the pointer in the thread environment block (TEB) that references the list of registered exception handlers.

The attacker needs to mock up a list entry as part of the payload and modify the first exception handler field using an arbitrary memory write.

System Default Exception Handling

Windows provides a global exception filter and handler for the entire process that is called if no previous exception handler can handle the exception.

Implement an unhandled exception filter for the entire process to gracefully handle unexpected error conditions and for debugging.

An unhandled exception filter function is assigned using the `SetUnhandledExceptionFilter()` function.

If an attacker overwrites specific memory addresses through an arbitrary memory write, the unhandled exception filter can be redirected to run arbitrary code.

Agenda

Pointer Subterfuge

Function Pointers

Data Pointers

Modifying the Instruction Pointer

Examples

Mitigation Strategies

Summary

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.

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.

Agenda

Pointer Subterfuge

Function Pointers

Data Pointers

Modifying the Instruction Pointer

Examples

Mitigation Strategies

Summary

Summary - 1

Buffer overflows can be used to overwrite function or data pointers in the same fashion a stack smashing attack is used to overwrite a return address.

The ability to overwrite a function or data pointer depends on the proximity of the buffer overflow to the target pointer, but targets of opportunity often exist in the same memory segment.

Clobbering a function pointer allows an attacker to directly transfer control to arbitrary, attacker-supplied code.

The ability to modify a data pointer and assigned value creates an arbitrary memory write.

Summary - 2

There are many opportunities for transferring control to arbitrary code given an arbitrary memory write.

Some of these targets are the result of standard features (e.g., longjmp(), atexit()).

Some are specific to particular compilers (e.g., .dtors section) or operating system (e.g., on_exit()).

Arbitrary memory writes can easily defeat canary-based protection schemes.

Summary - 3

Write-protecting targets is difficult because of the number of targets and because there is a requirement to modify many of these targets (e.g., function pointers) at runtime.

Buffer overflows occurring in any memory segment can be exploited to execute arbitrary code, so moving variables from the stack to the data segment or heap is not a solution.

The best approach to preventing pointer subterfuge resulting from buffer overflows is to eliminate possible buffer overflow conditions.