Chapter 12 - C++ Differences

Monday, July 31, 2023 3:34 PM

C++ Differences

1. Standard library
    a. Use of std::string removes a lot of potential memory corruption
    b. Std::cout<<msg<<std::endl makes for good obfuscation
2. Memory management
3. Exceptions
    a. On windows: SEH
        i. Structured exception handling
        ii. Is pwnable
        iii. https://www.securitysift.com/windows-exploit-development-part-6-seh-exploits/
4. Classes
    a. Usually used in C++
    b. Basic classes look like structs
    c. Inheritance
    d. Vtables
        i. Virtual table
        ii. A pointer to an array of function pointers
            1) Usually first (4|8) bytes of the class
            2) Pointers to virtual functions only
        iii. Exploitation
            1) Instance replacement(what if we change which Vtable is used)
            2) Function pointers are memory too

Chapter 13 - Linux Kernel Exploitation

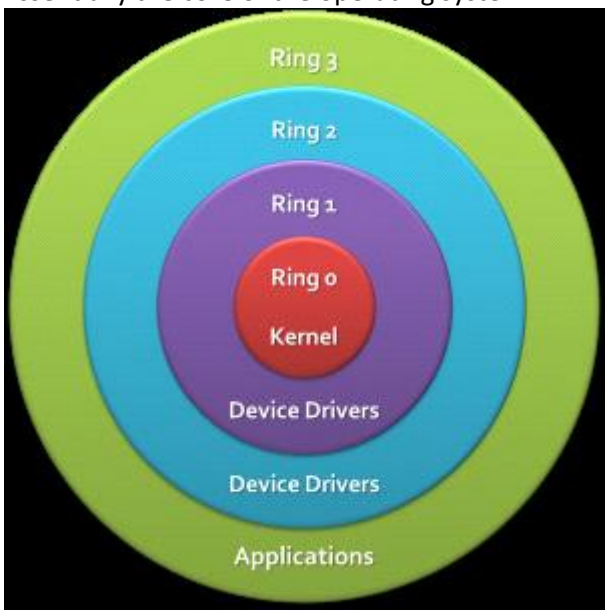
Monday, July 31, 2023 3:34 PM

Everthing up to now has been in the userspace. It is an abstraction that runs on top of the kernel

1. Filesystem I/O
2. Privilege Levels(Per User/Per Group)
3. Syscalls
4. Processes
5. And more…

What is a Kernel?

- Low level code with two major responsibilities
    - Interact with and control hardware components
    - Provide an environment in which applications can run
    - Essentially the core of the operating system



Ring 1 and Ring 2 are not utilized by most popular OS(Linux/Windows/OSX)

We've been working in Ring 3 so far

Keywords to look into: Remote Code Execution

Basics

- Managing your processes
- Managing your Memory
- Coordinating your hardware

Have to be able to go from a vulnerability to privileged execution without bringing down the system. Also need to be able to make it back to userland to make it easier

to execute whatever we might want to.

Types of expoitations
- Buffer overflows
- Signedness issues
- Partial overwrites
- Use-After-Free https://owasp.org/www-community/vulnerabilities/Using_freed_memory

First place to look is inside of the Loadable Kernel Modules(LKMs)
- Device drivers
- Filesystem Drivers
- Networking Drivers
- Executable interpreters
- Kernel extensions
- Useful commands that deal with LKMs
    - Insmod - Insert a module into the running kernel
    - Rmmod - Removes a module from the running kernel
    - Ismod - List currently loaded modules

Gaining Code Execution
- Printf() - printk()
- Memcpy() - copy_from_user()
- Malloc() - kmalloc()
- Debugging can be difficult
    - Rely on stack dumps, error messages, and other black box techniques

Use dmesg if the crash isn't fatal

The linux Kernel has a wrapper for updating process credentials

```
int commit_creds(struct cred *new) {



•••
}
```

Valid struct for creds

```
struct cred *prepare_kernel_cred(struct task_struct *daemon) {
…




}
```

Map of what you need to do

1. Create a “root” “struct creds” by calling prepare_kernel_cred(NULL);
2. Call commit_creds(root cred *);

3. Enjoy our new root privileges!

Easiest strategy for getting back to userland is high jacking execution and letting the kernel return by itself.

Kernel Space Protections

- MMAP_MIN_ADDR
    - Makes exploiting NULL pointer dereferences harder
      mmap_min_addr disallows
      programs from allocating low memory.

      Makes it much more difficult to exploit a simple NULL pointer dereference in the kernel.

- KALLSYMS
    - Gives the address of all symbols in the kernel(info leak)
    - Used to be world-readable but now it returns 0's for unprivileged users
- RANDSTACK
- STACKLEAK
- SMEP/SMAP
    - Supervisor Mode Execution Protection
    - Supervisor Mode Access Protection
    - Exploit is a Supply your own "get root" code

```
void get_r00t() {
commit_creds(prepare_kernel_cred(0));
}
int main(int argc, char * argv) {
…
trigger_fp_overwrite(&get_r00t);
…
//trigger fp use
trigger_vuln_fp();
// Kernel Executes get_r00t
...
// Now we have root
system(“/bin/sh”);
}
```

    - SMEP prevents this by triggering a page fault if the processor tries to execute memory that has the user bit set while in ring 0
    - Need to use ROP

Chapter 14 - x64, ARM, Windows

Monday, July 31, 2023 4:23 PM

X86

Calling conventions

- Cdecl
  - Caller cleans up the stack
  - Unknown or variable # of arguments. (ex: printf())
- Stdcall
  - Callee cleans up the stack
  - Standard calling convention for the Win32 API
- Fastcall
  - First two arguments are put into the ECX, and EDX, the rest are put onto the stack.

HAE x200

Tuesday, August 22, 2023 3:48 PM

Chapter 2: Programming

0x252 The x86 Processor

- EAX = Accumulator
- ECX = Counter
- EDX = Data
- EBX = Base
- ESP = Stack Pointer
- EBP = Base Pointer
- ESI = Source Index
- EDI = Destination Index
- EIP = Instruction Pointer

0x 260 Back to Basics

-

The Stack:

- The stack is a data structure used for function calls.
- It follows a Last-In-First-Out (LIFO) order.
- Function calls push data onto the stack.

Exploiting Buffer Overflows:

- Buffer overflows are common vulnerabilities.
- Occur when a program writes beyond the allocated memory.
- Can be exploited to inject malicious code.

Command-Line Arguments in C Programming

Introduction:

- Many non-graphical programs use command-line arguments for input.
- Command-line arguments are efficient and do not require user interaction.
- In C, command-line arguments are accessed through the main() function.

Accessing Command-Line Arguments:

- In C, use int main(int arg_count, char *arg_list[]) to access command-line arguments.
- arg_count holds the number of arguments, and arg_list is an array of strings.
- The zeroth argument is always the program's name, followed by other arguments.
- Example: commandline.c

```
#include <stdio.h>

int main(int arg_count, char *arg_list[]) {
  int i;
```

```
    printf("There were %d arguments provided:\n", arg_count);
    for (i = 0; i < arg_count; i++)
        printf("argument #%d\t-\t%s\n", i, arg_list[i]);
}
```

Using Command-Line Arguments as Integers:

- Command-line arguments are initially passed as strings.
- Use conversion functions like atoi() to convert strings to integers.
- Proper error checking is essential to avoid crashes

Example: convert.c

```
#include <stdio.h>

void usage(char *program_name) {
    printf("Usage: %s <message> <# of times to repeat>\n", program_name);
    exit(1);
}

int main(int argc, char *argv[]) {
    int i, count;
    if (argc < 3)
        usage(argv[0]);
    count = atoi(argv[2]);
    printf("Repeating %d times..\n", count);
    for (i = 0; i < count; i++)
        printf("%3d - %s\n", i, argv[1]);
}
```

Handling Insufficient Command-Line Arguments:

- Always check if there are enough command-line arguments before accessing them.
- Insufficient arguments can lead to segmentation faults.

Example: convert2.c

```
#include <stdio.h>

void usage(char *program_name) {
    printf("Usage: %s <message> <# of times to repeat>\n", program_name);
    exit(1);
}

int main(int argc, char *argv[]) {
    int i, count;
    // if(argc < 3)
    //     usage(argv[0]);
    count = atoi(argv[2]);
    printf("Repeating %d times..\n", count);
    for (i = 0; i < count; i++)
        printf("%3d - %s\n", i, argv[1]);
```

```
}
```

Exploring Segmentation Faults with GDB:

- Segmentation faults occur when accessing out-of-bounds memory.
- Use the GNU Debugger (GDB) to diagnose and debug segmentation faults.
- GDB provides backtraces and allows inspection of memory contents.

Variable Scoping in C Programming

Introduction:

- Variable scoping in C defines the context in which variables exist.
- Each function has its own set of local variables, independent of other functions.
- Global and static variables have broader scopes, persisting across function calls.

Local Variables and Function Context:

- Local variables exist within the context of a specific function.
- Multiple calls to the same function have their own independent set of local variables.
- Example: scope.c demonstrates nested function calls and local variable scoping.

Example: scope.c

```
#include <stdio.h>

void func3() {
   int i = 11;
   printf("\t\t\t[in func3] i = %d\n", i);
}

void func2() {
   int i = 7;
   printf("\t\t[in func2] i = %d\n", i);
   func3();
   printf("\t\t[back in func2] i = %d\n", i);
}

void func1() {
   int i = 5;
   printf("\t[in func1] i = %d\n", i);
   func2();
   printf("\t[back in func1] i = %d\n", i);
}

int main() {
   int i = 3;
   printf("[in main] i = %d\n", i);
   func1();
   printf("[back in main] i = %d\n", i);
}
```

Global Variables:

Global variables are declared outside of any function and are accessible from any function.
They persist across function calls and have a global scope.
Example: scope2.c demonstrates global variables.

Example: scope2.c

```
#include <stdio.h>

int j = 42; // Global variable

void func3() {
   int i = 11, j = 999; // Local variable in func3
   printf("\t\t\t[in func3] i = %d, j = %d\n", i, j);
}

void func2() {
   int i = 7;
   printf("\t\t[in func2] i = %d, j = %d\n", i, j);
   printf("\t\t[in func2] setting j = 1337\n");
   j = 1337; // Writing to the global j
   func3();
   printf("\t\t[back in func2] i = %d, j = %d\n", i, j);
}

void func1() {
   int i = 5;
   printf("\t[in func1] i = %d, j = %d\n", i, j);
   func2();
   printf("\t[back in func1] i = %d, j = %d\n", i, j);
}

int main() {
   int i = 3;
   printf("[in main] i = %d, j = %d\n", i, j);
   func1();
   printf("[back in main] i = %d, j = %d\n", i, j);
}
```

Static Variables:

- Static variables are declared with the static keyword.
- They have local scope within a function but retain their values between function calls.
- Example: static.c demonstrates static variables.

Example: static.c

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

```
void function() {
   int var = 5;
   static int static_var = 5; // Static variable initialization
   printf("\t[in function] var = %d\n", var);
   printf("\t[in function] static_var = %d\n", static_var);
   var++; // Add 1 to var.
   static_var++; // Add 1 to static_var.
}

int main() {
   int i;
   static int static_var = 1337; // Another static variable in main()
   for (i = 0; i < 5; i++) {
      printf("[in main] static_var = %d\n", static_var);
      function();
   }
}
```

Memory Segmentation in Compiled Programs:

A compiled program's memory is divided into five segments: text, data, bss, heap, and stack, each serving a specific purpose:

Text Segment (Code Segment):

- Contains the machine language instructions of the program.
- Code execution is nonlinear due to high-level control structures.
- EIP (instruction pointer) points to the first instruction in the text segment.
- The processor follows an execution loop reading, advancing, and executing instructions.
- Write permission is disabled to prevent modification.
- Read-only nature allows sharing among multiple program instances.

Data and bss Segments:

- Used to store global and static program variables.
- Data segment stores initialized variables.
- bss segment stores uninitialized variables.
- Both segments are writable but have fixed sizes.
- Global and static variables persist across function calls.

Heap Segment:

- Controlled by programmers.
- Memory allocation and deallocation using malloc() and free() functions.
- Size can grow or shrink as needed.
- Allocated memory managed by allocator and deallocator algorithms.
- Memory reservations can be made and released on the fly.
- Grows downward toward higher memory addresses.

Stack Segment:

- Variable-sized temporary scratch pad.

- Stores local function variables and context during function calls.
- Contains multiple stack frames.
- Implemented as a stack data structure with FILO (first-in, last-out) order.
- ESP register tracks the stack's end address.
- Stack grows upward in memory toward lower addresses.
- Stack frames store parameters, local variables, saved frame pointers (SFP), and return addresses.

Example: Stack and Memory Segments in C:

In C, memory segments play a crucial role in organizing variables:

- Global and static variables reside in the data and bss segments.
- The heap segment allows dynamic memory allocation using malloc() and free().
- The stack segment stores local function variables and contexts during function calls.

Memory_segments.c:

This program demonstrates memory segments in C:

- Global and static variables are in the data and bss segments.
- Heap memory is allocated and freed using malloc() and free().
- Stack variables show context separation within different functions.

Heap_example.c:

This program illustrates dynamic memory allocation using malloc() and free() functions. It allocates, deallocates, and reallocates memory on the heap, showcasing heap memory behavior.

Error-Checked malloc():

- To ensure robust code, error checks for malloc() calls can be centralized in a function (errorchecked_malloc()). This approach enhances code clarity and reduces redundancy when dealing with multiple memory allocations.

Key Takeaways:

- Programming is essential for hacking and exploitation.
- C is a valuable language due to its low-level control.
- Understanding memory management is critical for security research.
- GDB is a helpful tool for debugging and finding vulnerabilities.
- Shellcode plays a role in exploiting software vulnerabilities.
- Practice through programming challenges is essential for skill development.

GDB Cheat Sheet

Starting and Stopping GDB:

- gdb [executable] - Start GDB with the specified executable.
- quit or q - Exit GDB.

Running and Controlling Execution:

- run [args] or r - Start the program with optional arguments.
- continue or c - Continue program execution.
- next or n - Execute the next line (skipping function calls).
- step or s - Execute the next line (entering function calls).
- finish - Continue execution until the current function returns.
- break [location] or b - Set a breakpoint at the specified location.
- delete [breakpoint] - Delete a breakpoint.
- info breakpoints - List all breakpoints.
- disable [breakpoint] - Disable a breakpoint.
- enable [breakpoint] - Enable a disabled breakpoint.

Inspecting and Printing Values:

- print [expression] or p - Evaluate and print the value of an expression.
- info locals - List local variables in the current stack frame.
- info args - List function arguments in the current stack frame.
- backtrace or bt - Print a backtrace of the call stack.
- frame [num] - Select a specific stack frame.
- up and down - Move up and down the call stack.
- info registers - Display CPU registers.
- x/[format] [address] - Examine memory at a specific address using the specified format.

Manipulating Program State:

- set variable [name=value] - Set the value of a variable.
- set args [args] - Set the program arguments.
- set environment [var=value] - Set an environment variable.
- run - Restart the program with the new settings.
- catch [event] - Break when a specified event occurs (e.g., catch syscall).

Managing Threads:

- info threads - List all threads.
- thread [thread_num] - Switch to a specific thread.
- thread apply [thread_num] [command] - Execute a command in a specific thread.
- set scheduler-locking off - Disable thread locking (use with caution).

Working with Core Dumps:

- core-file [file] - Load a core dump file.
- generate-core-file - Generate a core dump file.
- info core - Display information about the core dump.

Debugging External Programs:

- attach [pid] - Attach GDB to a running process.
- detach - Detach GDB from the current process.

Scripting and Automation:

- source [script_file] - Execute GDB commands from a script file.
- define [command] - Define a custom GDB command.

HAE x300

Wednesday, August 23, 2023 3:47 PM

Generalized Exploit Techniques

- Execution of arbitrary code

Buffer Overflow Vulnerabilities:

- Definition: Buffer overflow vulnerabilities occur when data overflows from one memory buffer into another, often leading to unintended consequences and potential security threats.
- Common Attack Vector: Many cyberattacks exploit buffer overflow vulnerabilities to execute arbitrary code, compromise systems, or escalate privileges.
- Persistence: These vulnerabilities have persisted in software development for decades and remain a significant security concern.

The C Programming Language and Data Integrity:

- C Language: C is a widely-used high-level programming language known for its efficiency and flexibility.
- Data Integrity Responsibility: In C, programmers bear the responsibility of maintaining data integrity, as the language provides no built-in safeguards against buffer overflows.

Example: overflow_example.c (Buffer Overflow Illustration):

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

int main(int argc, char *argv[]) {
  int value = 5;
  char buffer_one[8], buffer_two[8];

  strcpy(buffer_one, "one");
  strcpy(buffer_two, "two");

  printf("[BEFORE] buffer_two is at %p and contains '%s'\n", buffer_two, buffer_two);
  printf("[BEFORE] buffer_one is at %p and contains '%s'\n", buffer_one, buffer_one);
  printf("[BEFORE] value is at %p and is %d (0x%08x)\n", &value, value, value);

  strcpy(buffer_two, argv[1]);

  printf("[AFTER] buffer_two is at %p and contains '%s'\n", buffer_two, buffer_two);
  printf("[AFTER] buffer_one is at %p and contains '%s'\n", buffer_one, buffer_one);
  printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value, value, value);
}
```

Buffer Overflow Scenario: When given a long command-line argument, this program demonstrates a buffer overflow, causing unexpected behavior.

Real-World Implications:

- Exploiting Buffer Overflows: In the wrong hands, buffer overflows can be exploited to take control of a program. An example is the provided exploit_notesearch.c, which gains root shell access.

Stack-Based Buffer Overflow Vulnerabilities:

- Definition: Stack-based buffer overflow vulnerabilities occur when data overflows from one stack frame to another, potentially altering program execution flow.

Example: auth_overflow.c (Stack-Based Buffer Overflow):

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

int check_authentication(char *password) {
   int auth_flag = 0;
   char password_buffer[16];

   strcpy(password_buffer, password);

   if (strcmp(password_buffer, "brillig") == 0)
      auth_flag = 1;
   if (strcmp(password_buffer, "outgrabe") == 0)
      auth_flag = 1;

   return auth_flag;
}

int main(int argc, char *argv[]) {
   if (argc < 2) {
      printf("Usage: %s <password>\n", argv[0]);
      return 1;
   }

   if (check_authentication(argv[1])) {
      printf("Access granted.\n");
   } else {
      printf("Access denied.\n");
   }

   return 0;
}
```

Buffer Overflow Scenario: By providing a long input, this program demonstrates a stack-based buffer overflow, which results in unintended access being granted.

Setting Environment Variables:

- Environment variables can be used for storing data or shellcode in memory for exploitation.
- To set an environment variable, use the export command, followed by the variable name and its value.

- Accessing Environment Variables:

Environment variable values can be accessed by prefixing their names with a dollar sign, like $MYVAR. The env command displays a list of all environment variables currently set.

Preparing Shellcode:

- Shellcode is binary code used in exploits, which needs to be in binary form for execution.
- The example code exploit_notesearch.c contains sample shellcode that should be converted to binary form.

Extracting Hex-Expanded Shellcode:

- The head command is used to display the first few lines of the exploit_notesearch.c code.
- grep with the pattern ^\" isolates lines starting with a quotation mark, containing the shellcode.
- cut is used to extract the bytes between the quotation marks.

Conversion to Binary:

- A loop processes each line of shellcode and converts it to binary form.
- The echo -en command prints the hex-expanded shellcode without adding a newline character.
- The resulting binary shellcode is saved to a file named shellcode.bin.

Viewing Binary Shellcode:

- hexdump -C shellcode.bin displays the contents of shellcode.bin in hexadecimal format.

Storing Shellcode in an Environment Variable:

- Binary shellcode is stored in an environment variable named SHELLCODE, with a generous NOP sled prepended.
- This approach ensures the shellcode is available in the environment for later exploitation.

Determining the Address of the Shellcode:

- The GDB debugger is used to examine memory near the bottom of the stack in the notesearch program.
- i r esp command shows the current value of the ESP (stack pointer) register.
- x/24s $esp + 0x240 displays 24 null-terminated strings from the memory location near the ESP.
- The address where the shellcode is stored is identified as the target address for exploitation.

Exploiting with Environment Variables:

- The exploit_notesearch_env.c code demonstrates how to exploit the notesearch program using environment variables.
- It creates an array env containing only the shellcode as the environment variable.
- Calculates the target address accurately, considering the length of the program name and shellcode size.
- Allocates a buffer to hold the return address and fills it with the calculated target address.
- Finally, uses execle() to execute the notesearch program with the provided buffer and environment variable.

Execution and Exploitation:

- The code is compiled with gcc to create the exploit_notesearch_env executable.
- When executed, the program overflows the return address with the calculated shellcode address, triggering the exploitation.
- The notesearch program runs with the provided shellcode, leading to executing arbitrary commands.
- The example demonstrates running the whoami command, resulting in root access.

Exploring Environment Variable Address Predictability:

- The getenvaddr.c program calculates the exact address of an environment variable during the execution of a target program.
- It adjusts the address based on the difference in program name length, providing an accurate prediction.
- This eliminates the need for a NOP sled, making the exploit more precise and reliable.

Usage of execle() Over system():

- The system() function is used to execute commands, but it starts a new process and may not transfer privileges.
- execle() is a more precise way to run commands by replacing the current process, and it allows specifying the environment.
- The example code exploit_notesearch_env.c utilizes execle() for exploitation, eliminating the need for a NOP sled and starting additional processes.

Analyzing the Buffer Overflow:

- Debugger (GDB): The text suggests using a debugger like GDB to examine the program's memory during execution.
- Setting Breakpoints: Breakpoints can be set to inspect stack frames and observe the impact of the buffer overflow.

0x341 A Basic Heap-Based Overflow

- Introduction to a buffer overflow vulnerability in the notetaker program.
- Allocation of two buffers, buffer and datafile, on the heap using ec_malloc().
- Vulnerability when user input exceeds the buffer size, overflowing into adjacent memory.
- Demonstrated by providing 104 bytes of input to overflow buffer, affecting datafile.
- Error handling issues triggered when freeing heap memory.
- Discussion on exploiting the vulnerability to append entries to system files like /etc/passwd.
- Use of symbolic links to ensure validity of appended entries.
- Crafting specific strings to exploit the heap overflow vulnerability for privilege escalation.

0x342 Overflowing Function Pointers

- Discussion of the game_of_chance.c program with a function pointer vulnerability.
- Introduction of the current_game function pointer within the player structure.
- Identification of a potential buffer overflow vulnerability in the input_name() function.
- Explanation of how overflowing the name buffer can overwrite the current_game function pointer.
- Need for a game to be played first to set the last_game variable for function pointer execution.

- Selection of the jackpot() function as a suitable target for exploitation.
- Exploiting the vulnerability by overwriting the current_game pointer with the jackpot() function's address.
- Automation of a sequence of menu selections to trigger the buffer overflow and execute the jackpot() function.