Name: Vedant Deepak Borkute
SID: 862552981
NetID: vbork001
HW 2

Smashing The Stack For Fun And Profit :

1. What is NOP sledge and why it is useful?

A NOP sledge is a padding of NOP instructions (0x90 in the Intel architecture) placed at the front of the overflow buffer. It is useful because it helps a successful exploit. If the overwritten return address points anywhere in the NOP string, the NOPs will execute until control flows into the actual shellcode.

2. Why the final shellcode uses xorl %eax,%eax instead of moving 0 to register?

The shellcode uses xorl %eax,%eax to write a zero value (0x0) into the register because null bytes (\x00) in the shellcode string are problematic. If a buffer copying function (like strcpy) encounters a null byte, it treats it as the end of the string and stops copying, thus the full shellcode wont be injected into the buffer.

3. What is the simplest way to find potential buffer overflow vulnerabilities?

We use grep on the source code to find functions that perform no boundary checking like strcat(), strcpy(), sprintf(), vsprintf(), and gets().

……..

StackGuard:

1. How did it mitigate the easy-to-guess canary problem?

StackGuard mitigated the easy-to-guess canary problem by using randomly chosen canary values. It modified the crt0 library to choose a set of distinct random canary words when the program starts, making it difficult for an attacker to predict the value.

2. MemGuard provides better protection than StackGuard, why it is not widely adopted like StackGuard?

MemGuard imposes substantial performance penalties, as it requires heavy-weight system calls to access privileged hardware, such as the Pentium's debugging registers. Microbenchmarks showed function calls slowing down by 70x or 160x, making it generally unsuitable for production use as compared to StackGuard.

……..

From the ROP (return-oriented programming) :

1. Why does the Galileo algorithm scans backwards instead of forwards?

The Galileo algorithm scans backwards because it is far simpler than disassembling forwards from every possible memory location. By starting from an instruction already found to be a ret instruction, the algorithm ensures that the instruction sequence found so far is the suffix for all further to be found sequences.

2. Explain Figure 10, why it's an infinite loop?

In fig 10 we have a gadget used for an unconditional jump, consisting of the address of the sequence pop %esp; ret followed by the gadget's own address on the stack. The pop %esp instruction loads the gadget's starting address into the stack pointer (%esp). The subsequent ret instruction uses this new %esp value to fetch the next instruction, thereby jumping back to the beginning of the gadget and creating an infinite loop.

3. ROPgadget is a tool to facilitate ROP exploits, try its --ropchain option on a libc binary (e.g., /lib/x86_64-linux-gnu/libc.so.6) and explain how the found shellcode works, referring to Section 4 of the ROP paper.

A return-oriented shellcode is constructed as a chain of gadgets placed on the stack. The example shellcode invokes the execve system call to execute /bin/sh. It uses a sequence of load/store and arithmetic gadgets to load the system call index (0xb) and arguments (the path /bin/sh, argv, and envp pointers) into the appropriate registers (%eax, %ebx, %ecx, and %edx). Once the registers are configured, the chain terminates with a sequence that traps into the kernel, such as an lcall instruction, thereby initiating the system call.

……..

From the Control-flow Integrity (CFI):

1. What are the three requirements/assumptions for CFI enforcement? The three critical assumptions for CFI enforcement are:

    a. UNQ (Unique IDs): Bit patterns chosen as IDs must not be present elsewhere in code memory, except in IDs and ID-checks.
    b. NWC (Nonwritable Code): The program must not be able to modify code memory at runtime.
    c. NXD (Nonexecutable Data): Data memory must not be executable as code.

2. From the CFI paper, what is the strategy mentioned in the paper that can improve the accuracy of CFG?

The strategy mentioned for improving CFG accuracy is code duplication. Code duplication involves creating separate copies of functions so that they can target different destination sets upon return, which can eliminate the possibility of overlapping yet distinct destination sets.

……

From AddressSanitizer:

1. How does ASAN detect spatial memory errors like buffer overflow?

   ASAN detects spatial memory errors, such as out-of-bounds accesses to heap, stack, and global objects, primarily through the use of poisoned redzones and shadow memory. The instrumentation module modifies code to check the shadow state for each memory access and creates these unaddressable redzones around stack and global objects. The specialized run-time library, which replaces standard memory allocation functions, creates poisoned redzones around allocated heap regions. If an instrumented memory access attempts to read or write into a poisoned redzone, an out-of-bounds error is detected and reported.

2. How does ASAN detect temporal memory errors like use-after-free?

   ASAN detects use-after-free bugs using its specialized run-time library, which manages shadow memory and replaces memory allocation functions (malloc, free). When memory is freed via free, the function poisons the entire memory region (marks it as unaddressable) and places it into quarantine. By holding the freed region in quarantine, ASAN delays its reuse. If the application later accesses this memory while it remains poisoned, the shadow memory check performed by the instrumentation detects the temporal error.

…….