CS2107 Live Class Lec7: Programming Security

$_{/}|/|U_Ch@NgRu!$

May 11, 2021

1 Overview

1.1 Requirement

1. correct
2. efficient
3. secure

But often, program behaves beyond its intended behavior

1.2 Vulnerability

1. Insecure implementation(allowing attacker to deviate from the programmer's intents
2. Input that's not been considered before; The attacker gives some input that is not anticipated before, the input may let the attacker
   (a) Access sensitive resources
   (b) Execute injected code
   (c) Deviate from the original intended execution path

2 Buffer Over flow

2.1 Some example

1. Morris worm(1988): Exploited a UNIX finger service to propagate itself over the Internet
2. CodeRed worm(2001):Exploited Microsoft' s IIS 5.0
3. SQLSlammerworm(2003): Compromised machines running Microsoft SQL Server 2000

4. Various attacks on game consoles such that unlicensed software can run without the need for hardware modifications
    - Xbox
    - PlayStation 2(PS2 Independence Exploit)
    - Wii(Twilight Hack)

3 SQL Injection

3.1 Examples

1. Yahoo!(2012):450,000 login credentials claimed to be stolen using a "union-based SQL injection technique"
2. British Telco TalkTalk(2015): A vulnerability in a legacy web portal was exploited to steal the personal details of 156959 customers

4 Integer Overflow

1. EuropeanSpaceAgency' sAriane5Rocket(1996):Unhandled arithmetic overflow in the engine steering software caused its crash, costing $7 billion
2. ResortsWorldCasino(2016): A casino machine printed a prize ticket of $42,949,672.76.

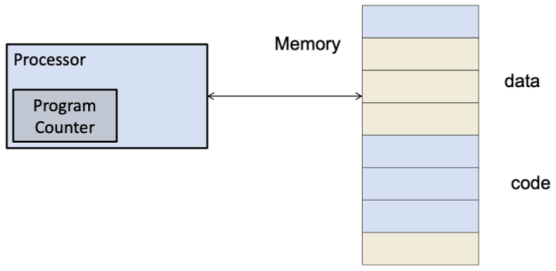
5 The reason of so many vulnerability

1. Functionality is the first goal during design and implementation, while security is usually placed at the second(the feathers pay the bills)
2. Unavoidable human mistakes
    - Lack aware of secure problems
    - Careless programmer
3. Complex modern computing system, some current systems are very complex(Windowx XP has 45 millions codes, which leaves big attack surface)
4. Learn programming language fast(some websites help learn the programming fast, but the programmer might not be aware of the implicit change and the bellow implementation of the language(e.g. the random in javascript)

6 Vulnerability Code and Data in Mordern computer

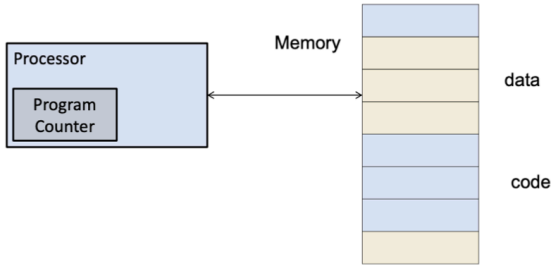
Modern computers are based on the Von Neumann computer architecture, which means code and data are stored together in the memory and there is no clear distingctionn between code and data The key factor is that Program may be tricked into treating input data as code(Basis for code-injection attacks)

6.1 Control Flow and program Counter



The processor, as the name suggests, is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logic controlling and the I/O operations specified by the instructions
These instructions are read with the help of program counter, or instruction pointer/cpunter. The program counter is a processor register that indicate the address of the next instruction
After an instruction is completed. the processor fetches the next instruction from the address stored in the program counter. Once this fetching is done, the program counter changes to the next position

6.1.1 Change to Program Counter



1. Increment
2. Direct jump: replace with a constant value specified in the instruction

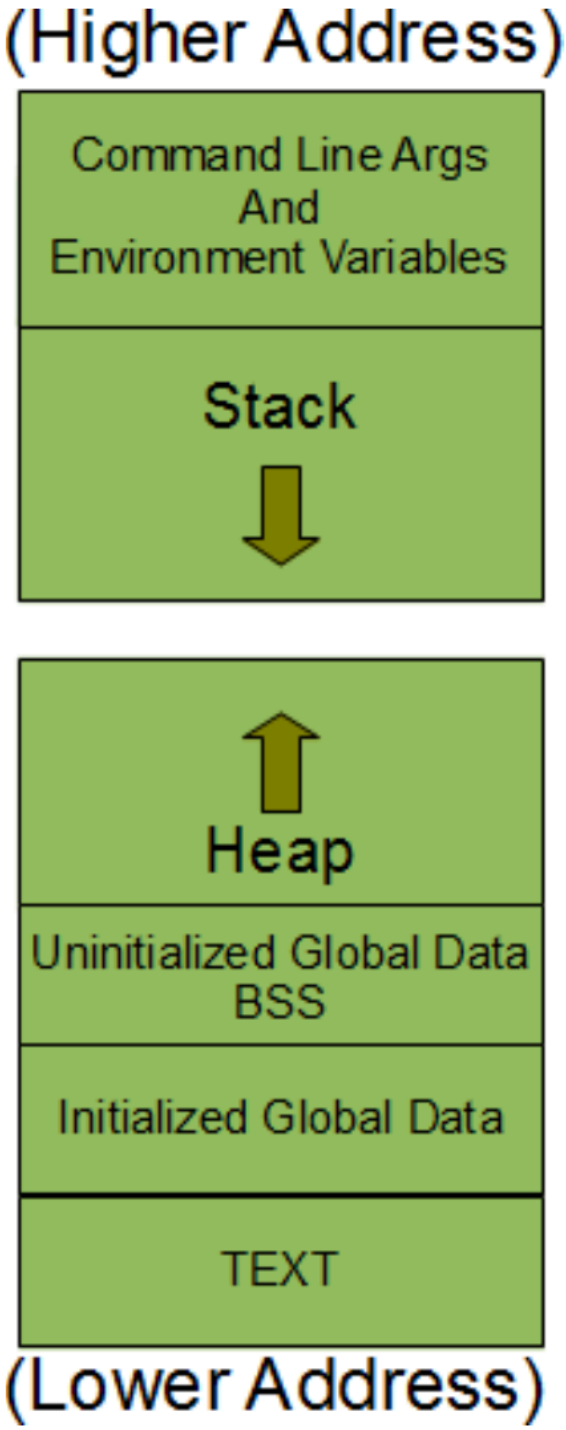
3. Indirect jump: replaced with a value fetched from memory; There are many different forms of indirect jump

6.2 Stack

Also called execution stack or Call stack

6.2.1 Functions

Functions break code into smaller pieces and facilitate modular design and code reuse. A function can be called in many different locations, and can be called may times.(e.g. recursive function)



So the Linux process is like the above

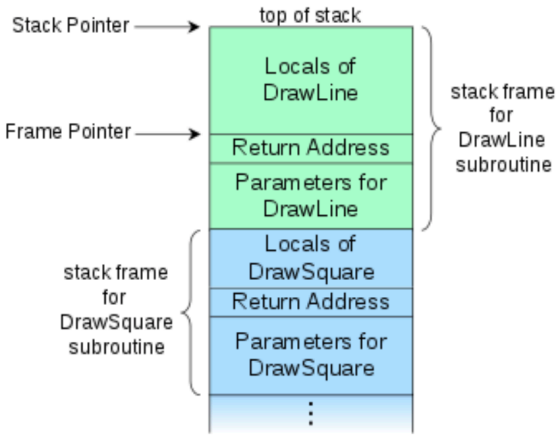
- Command Line Arguments and Environment Variables(Stored at the top of the process memory layout at the higher address
- Stack: Memory area used by the process to store local variables of functions and other information that saved every time a function is called. More about the later
  Grows downwards
- Heap:
  - Used for dynamic memory allocation
  - Shared among all threads of a single process, but not between different processes running in the system
  - Memory from this segment should be used cautiously and should be deallocated as soon as the process is done using that memory
  - Grows upwards
- Uninitialized Global Variables: They are initialized with the value zero; BSS stands for "Block Started by Symbol"
- Initialized Global Variable
- Text:
  - Memory area that contains the machine instructions that the CPU executes
  - Shared across different instances of the same program being executes
  - Since there is no point in changing the CPU instructions, this segment has read-only privileges

6.3 Call Stack

The Call stack is a data structure in the memory that stores important information of the running process. As with a stack, it is LIFO or FILO
In this module, the stack grows from high address to low address
The stack helps to keep track of

1. Control Flow information, such as return address
2. Parameters passed to functions
3. Local variables of the function

Each process pushes a stack frame to the call stack



As shown in the image above, the stack frame usually contains :
Higher address

- Passed parameters
- Return address back to caller
- Previous frame pointer or pointer to the previous stack frame
- Space for local variabl of the routine

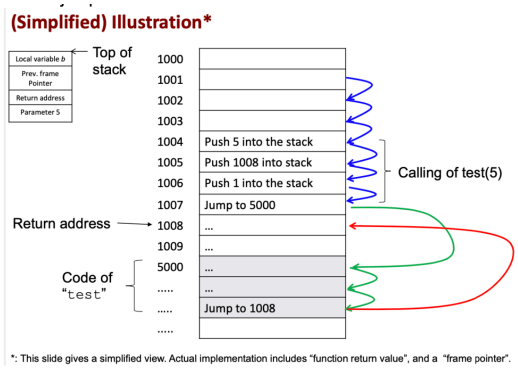
Frame pointer: a pointer within the stack frame that is unchanging. Unlike the stack pointer and the top of the stack that changes as local values are pushed and pop, the frame pointer indicates the start of the stack frame and can allow for easier access to variables via an offset.

6.3.1 A general example of the step flow

Consider a function test(int a) that declare a local variabl int b=1, the following happends:

1. Stack pushing:
   (a) Parameter 5 is pushed into the stack
   (b) Return address is pushed into the stack
   (c) Previous frame/stack pointers are pushed into the stack
   (d) Local variable b and its value 1 is pushed into the stack
2. Control flow jumps into the code of test
3. Execute fast
4. After test is completed the stack frame is popped from the stack

5. Control flow jumps to the caller(return address)



7 Control Flow Integrity

The call stack stores a return address, the location of a to-be-executed instruction, and data in the memory. An attacker can compromise a process's execution by either modifying the code or the control flow. It is difficult for the system to distinguish these malicious pieces of code from normal code

8 Memory Integrity

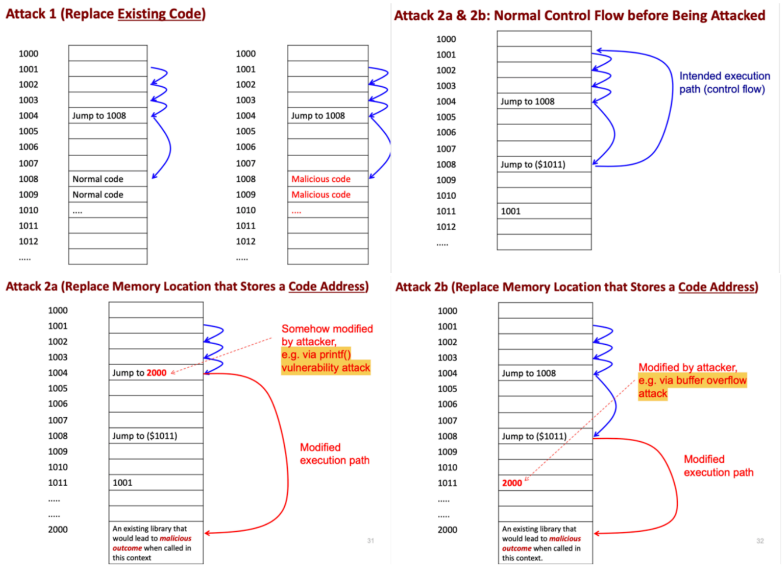
In general, it is not easy for an attacker to compromise memory integrity, i.e. modify data in the memory. Onw way an attacker can do that is to exploit some vulnerabilities so as to "trick" the victim process to write some of its memory locations, e.g. via buffer overflow attack. The above mechanisms typically have some restrictionsL

- The attacker can only write to a small amount of memory
- they can only write a sequence of consecutive bytes

9 Possible Attack Mechanism

Assuming that the attacker can now write to some memory ;locations, the attackers could:

1. Overwrite existing execution code portion with malicious code
2. Overwrite a piece of control-flow information
   (a) Replace a memory location storing a code address that is used by a direct jump
   (b) replace a memory location storing a code address that is used by a indirect jump



10 Attacks and Vulnerabilities

10.1 printf() and format string vulnerability

printf() is the C function for formatting output.
What is special about the printf() is that it can take in **any** number of arguments
The general format is as such:
int printf(const char * format, ...)
where ... is any amount of arguments that would be printed based on how the format string is written

10.1.1 Format specifiers

In the format string, there would be format specifiers to indicate the type of the variaable to be printed(printf("%d",IntNum);
Some common format specifiers:

- %d: decimal(int)
- %u: unsigned decimal (unsigned int)
- %x: hexadecimal (unsigned int)
- %s: string((const)(unsigned)char*)

- %n: number of bytes written so far, (*int)
- %p: pointer

Note that %s and %n are passed as reference

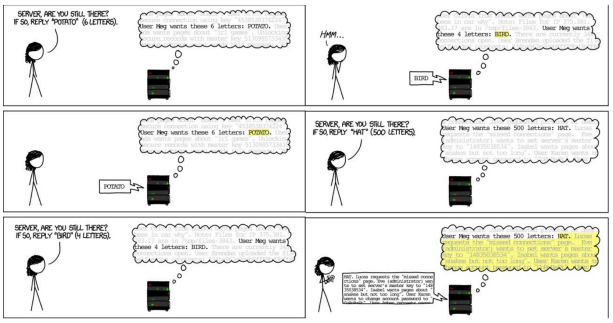
10.1.2 Missing arguments

When only one parameter is supplied, only that parameter will be displayed
printf("hello world");

- If the parameter is a string by itself: ok
- If it has a format specifier within, printf() will actually search for the second parameter in the stack to be displayed

The attacker can carefully design the string to be printed, so that:

1. Obtain more information of the program's call stack
   - Using "%d.%d.%d" to fetch values from the top and print them out as decimal values
     - sample output: 73896.0.269401708
   - Using "%08x.%08x.%08x" to fetch values from the top of the stack and print them out as lower-case hexadecimal, with 8 digits shown(from the 8, and with padding/prefixes of 0s(from the 0)
2. Cause the program to crash
   - "%s" will fetch a number from the stack and treat this number as an address
   - The printf() function will try to print out the memory contents pointed by this address as a string, until a null character is read
3. Big Picture Exploitation
   This vulnerability can be exploited:
   - In a multi-user setting
     - If the program has an elevated privilege i.e. via set-UID, an attacker may be able to obtain system-level information
   - In a client-server setting
     - If the server program is vulnerable, the attacker may be able to submit a request and obtain sensitve information



Preventive Measures: we should not take the user's input as a format string. Instead, we should read the input into seperate variable then print out that variable via our own format string. For example

- printf(t): the t is supported directly by users and the format will follow what the user input, which is insecure
- printf(f,t): the format is made by us, so it's more secure

In GNU Compiler Collection, the compiler will warning user for dangerous or suspect formats, the relevant flags are -Wall(enables warning)

11 Data Representation Security

Different part of a program or system may use different data representations, such inconsistency may lead to vulnerability
For example, CVE-2013-4073: "Ruby' s SSL client implements hostname identity check, but it does not properly handle hostnames in the certificate that contain null bytes."

11.1 String

In C, the printf() does not check the length of a string, but read until the first null byte(\0)
$However, not all system adopts such representation, there are two types$:

null-termination representation

non-null-termination representation

Exploitable Vulnerability:

1. NULL-Byte Injection: A certificate Authority may accept a null characters containing NULL bytes, e.g luminus.nus.edu.sg\0.*attacker.com* A verifier who uses both of the above representation conventions to verify the certificate could be vulnerable(by pass the filter which

cannot identify the part after \0