Buffer overflows

- Buffer overflow problem
- Memory layout
- Protection against buffer overflows
- Stack protection
- Other overflow attacks
- Dangerous functions in C
Buffer overflow problem

- Buffer overflow — one of the most widespread security problems in programs
- Common to low-level programming languages with manual memory management
- Morris’ Internet Worm in 1988 used buffer overflow as one of the exploits
- Lazy programmer does not check whether user data really fits into allocated memory area
- On attack the memory after allocated buffer is overwritten with attackers data
- As a result, other variables or control data are manipulated
Exploiting buffer overflows

- Von Neumann architecture of programmable computers: program and data are stored in the same way in memory
- Thus the attacker can provide his own memory image and trick the system into executing it as a program
- Buffer is usually either in heap or in stack (among data, not among code)
- With any buffer overflow, attacker can just change some variables in simplest case (without running his code — might or might not be exploitable)
Exploiting heap buffer overflows

• Heap is the area where runtime memory allocation takes memory from

• Often a doubly-linked list of memory areas

• Just data, no executable code

• Exploits overwriting object and virtual method pointers

• Exploits overwriting memory after buffer can influence next and prev pointers of next blocks, not their own

• Freeing memory after such corruption changes data pointed by tainted pointers, gain of control is possible

• Attacker has to know the memory allocator used
Exploiting stack buffer overflows

- Stack contains the data about function calls — return address, saved data, function parameters, local variables of called functions

- With buffers in stack, return address of the subroutine is overwritten and on returning from this subroutine a jump is made into attack code

- Attack code can be provided in overwritten memory and pointing return address to it makes the code run on returning from function

- Even 1-byte buffer overflow has been exploited!
  - Overwriting 1 byte of saved EBP stack frame pointer
Stack and buffers

Stack growth

Top of stack
Return address
Local variables
Buffer

Address increases

Top of stack
Attack code
Return address
Local variables
Buffer
Memory layout

- Stack frame for each level of subroutine call:
  - Caller-saved registers
  - Function arguments
  - Return address
  - Callee-saved registers
  - Local variables (automatically deallocated by function return)

- Exploit needs to know the exact address to return to
  - Can be alleviated with a block of NOP’s at the start of exploit code
Protection against stack overflows

• Real defence — fix the bugs
  – Check the length of all buffer writes
  – Do not use insecure library functions that do not check the length

• Only real defence can solve the problems for real but some other methods can help against exploiting of yet-undiscovered buffer overflows
Protection against stack overflows

- Memory protection (nonexecutable stack + heap, W\^X, \ldots)
  - Needs hardware support for the protection bits
  - NX bit on x86 CPUs
  - Segment overlap tricks for older x86 CPUs
- Inserting canary values into stack by compilers (StackGuard, ProPolice, StackGhost, \ldots)
  - Randomization
  - Null byte in the word
- Address space randomization
- Libraries that replace inherently insecure functions with variants that check for stack pointer (libsafe, libverify, \ldots)
### Stack protection with canary value

<table>
<thead>
<tr>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local variables</td>
</tr>
<tr>
<td>Canary value</td>
</tr>
<tr>
<td>Return address</td>
</tr>
<tr>
<td>Top of stack</td>
</tr>
</tbody>
</table>
Thwarting the protection

- Return into libc and other libraries
- Putting the exploit code together from other library fragments with specially constructed return chain
- Finding some unprotected (executable) memory areas
- Changing memory protection with other code snippets first
Some dangerous functions in C

strcpy(char *dest, const char *src)
  dest overflows
     → strncpy, strlcpy

strcat(char *dest, const char *src)
  dest overflows
     → strncat, strlcat

getwd(char *buf)
  buf overflows
     → getcwd

gets(char *s)
  s overflows
     → fgets
Dangerous I/O functions in C

\[\text{[vf]scanf(const char *format, ...)}\]

argument overflow

→ put argument lengths into the pattern

\[\text{[v]sprintf(char *str, const char *format, ...)}\]

str overflows

→ \[\text{[v]snprintf}\]