Chapter 1

Buffer Overflow

1.1 Stack and Heap

The *stack* is the memory set aside as scratch space for a thread of execution. When a function is called, a block is reserved on the top of the stack for local variables and some book keeping data. When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack; freeing a block from the stack is nothing more than adjusting one pointer.

The *heap* is memory set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time; there are many custom heap allocators available to tune heap performance for different usage patterns.

Each thread gets a stack, while there's typically only one heap for the application (although it isn't uncommon to have multiple heaps for different types of allocation).

The OS allocates the stack for each system-level thread when the thread is created. Typically the OS is called by the language runtime to allocate the heap for the application.

Scope: the stack is attached to a thread, so when the thread exits the stack is reclaimed. The heap is typically allocated at application startup by the runtime, and is reclaimed when the application (technically process) exits.

Size: the size of the stack is set when a thread is created. The size of the heap is set on application startup, but can grow as space is needed (the allocator requests more memory from the operating system).

Speed: the stack is faster because the access pattern makes it trivial to allocate and deallocate memory from it (a pointer/integer is simply incremented or decremented), while the heap has much more complex bookkeeping involved in an allocation or free. Also, each byte in the stack tends to be reused very frequently which means it tends to be mapped to the processor's cache, making it very fast. Another performance hit for the heap is that the heap, being mostly a global resource, typically has to be multi-threading safe, i.e. each allocation and deallocation needs to be - typically - synchronized with "all" other heap accesses in the program.

1.2 Call stack

EBP: it is a register pointing to the highest address of the current invocation frame
ESP: it is a register pointing to the top of the stack (lowest address)
EIP: it is a register pointing to the instruction to be executed.

1.3 Stack smashing

The canonical method for exploiting a stack based buffer overflow is to overwrite the function return address with a pointer to attacker-controlled data (usually on the stack itself).

1.4 Fixing buffer overflow

- Used counted versions of string functions (less performance but there is a boundary check)
- Use safe string libraries, if available, or C++ strings
  - fgets instead of gets
  - snprintf instead of printf
- Check loop termination and array boundaries
- Use C++/STL containers instead of C arrays

1.4.1 Non executable stack

The OS kernel can be patched so as to forbid the execution of instructions whose address is on the stack.

- functions trampolines are addede to the stack by gcc for nested functions
- signals handlers are allocated on the user stack by Linux
- dynamically generated code bmay reside on the stack

1.4.2 Canary words

Canaries or canary words are known values that are placed between a buffer and control data on the stack to monitor buffer overflows. When the buffer overflows, the first data to be corrupted will usually be the canary, and a failed verification of the canary data is therefore an alert of an overflow, which can then be handled, for example, by invalidating the corrupted data.

Chapter 2

Injecton

2.1 SQL injection

Problem. user provided data is used to form an SQL query (eg. through string concatenation) and the attacker provides malformed data aimed at changing the semantics of the query.
Controls: regular expressions, prepared statements, grant access to table only through stored procedures

2.2 Command injection

Problem. untrusted user data include commands as interpreter understands, forcing the interpreter to operate beyond its intended functions.
Controls:

- Deny-list: user data including characters ina deny list are rejected
- Allow-list: only user data matching the character patterns in the allow list are interpreted
- Quoting: user data transformed (eg. embedded with quotes) to avoid being interpreted as commands

2.3 Error handling

Example: the attacker provides an invalid file name hence a null pointer is used to perform file operations causing either DoS or disclosure of program and system's internals.
Controls:

- Handle error exceptions in the code
- Never mask exceptions that may corrupt the program's state
- Check function return values for errors

2.4 XSS

Problem. user input is directly displayed in an output webpage without any sanitization. the typical attack pattern is:

1. Identify a vulnerable website
2. Create a URL that submits malicious input
3. Social engineering to induce the victim to click on the URL
4. Victim clicks

Controls: HTML encoding, sanitize user input

Chapter 3

Network

3.1 SSL and TLS

Problems:

- The certificate authority signing the certificate is not validated
- The signature of the certification authority is not validated for integrity
- The time validity of the certificate is not checked
- The certificate is not checked against the certificate revocation list.

While authentication checks are mandatory in the https protocol, if programmers use low level SSL/TLS libraries directly they might forget some important authentication check.

3.2 Weak passwords

Problems:

- Initial passsword is weak
- Long passwords are not allowed
- Short password are allowed
- Dictionary passwords accepted
- Alphanumeric-only passwords accepted
- No checks on number of login attempts
- User is locked out when too many login attempts are made (DoS)
- user is not requested to change password periodically
- Previous passwords are allowed when user changes password
- No secure channel/protocol is used

- Change passwords are not reauthenticated
- Passwords can be reset upon end user request based on weak information
- Reset passwords are specified by the user rather than being delivered securely
- Default password change is not enforced on first login
- Passwords are stored in clear
- Login failure messages and response time reveal sensitive information
- Upon login failure wrong passwords are logged

Controls:

- Use PBKDF2 to hash passwords
- Iterate password hashing to increase response time
- Raise barriers to password reset (questions, secure delivery,extra authentication)
- Instead of locking out users let them retry after some time has passed
- Increase the response time when login failures occur
- Blacklist IP addresses (and alert the user) if too many login attempts occur
- Multifactor authentication: password + token