Buffer overflow and stack smashing attacks

Principles of application software security

Buffer overflows

- One of the most common vulnerabilities in software

- Particularly problematic when present in system libraries and other code that runs with high execution privileges.
How it works

- Application reserves adjacent memory locations (buffer) to store arguments to a function, or variable values.
- Attacker gives an argument too long to fit in the buffer.
- The application copies the whole argument, overflowing the buffer and overwriting memory space.
- If the conditions are “just right” this will enable attacker to **gain control over the program flow** and **execute arbitrary code**, with the same privileges of the original application.

Stack smashing

- Function (sub-routine) calls results in an **activation frame** being pushed onto a memory area called the **stack**.
Memory management

- The stack, which contains activation frames, starts at the highest memory address allocated for the process, and grows down.
- Variable-length data (say strings) that are read dynamically, are kept in the heap, which grows up.

How to smash

- Give the application a very long string with malicious code. The string overflows onto the stack and overwrites the return address.
- The return address now points to the beginning of the malicious code.
Canary Guards

- Like the legendary canary-in-the-mine, it detects stack smash attacks.

- Inserts a “Canary value” just below the return address (Stack Guard) or just below the previous frame pointer (Stack Smashing Protector). This value gets checked right before a function returns.

SSP

- Prevents overflow of local non-buffer variables

- Canary value checking only takes place at return time, so other attacks possible.
Alternatives to canaries

- Use a compiler that does full bounds checking, i.e., makes sure that the code always allocate enough memory for arguments.
  - Like SSP, code has to be re-compiled with this compiler
  - Significant performance penalty (Java/C)

Static analysis

- Use a code analyzer to detect buffer overflows
  - Since checking that arbitrary code does not overflow is an un-decidable problem, the code must be annotated in order for this to work
  - Advantage is that the compiled code does not suffer from performance deterioration
Safe libraries

- Many vulnerabilities in code are due to unsafe use of system libraries.
- An alternative is to install a kernel patch that dynamically substitutes calls to unsafe library functions for safe versions of those.
- Not possible for closed-source systems such as MS operating systems.

Memory address randomization

- Patch at the kernel level, changing the memory mapping.
- Small performance penalty, by extra memory lookups (actually, extra cache lookups)
- Makes it nearly impossible to perform a useful buffer overflow