# Binary Exploitation

## Categories:

* Stack overflow
* Format string vulnerability
* Heap overflow
* Memory Corruption (use after free, double free)
* Type confusion

Stack overflow:

* Uses out of bounds read/write
* Local variables located on the stack
  + Checks
  + Pointers
  + Other stuff
* Return Pointer and Base (stack) pointer located on the stack
* Stack pointed to by RSP

Format string vulnerability:

* Present if user supplied string is used as a format string for printf, sprint etc.
* Can be used as arbitrary read and arbitrary write (lead canaries)
* Controlled using format characters:
  + % - string
  + %p – pointer
  + %7%o99x – 7th argument taken from the stack printed as a hex value and extended to 99 characters (leaks anything you want, because there are not normally 7 values passes to the stack)
  + %n – Write the numbers of characters written so far to a given location
* Often both the buffer and data come from the stack – self feeding addresses

Heap overflow:

* Use out of bounds write to modify pointers in other head segments
* Can lead to head allocation to/from controlled location and other issues

Type confusions

* Unions

Memory corruptions

* Heap magic
  + Heap sprays
  + Holes
  + Buckets

Mitigations

* Address space layout randomization (ASLR)
* Position independent code
* Dynamix linking
* NX bit +Data execution prevention (DEP)
* Stack canaries
* Shadow stack
* Fortify source
* Sandboxing

Address space layout randomization:

* Most of the sections in the binary are placed on a random location in a memory
* Some sections might be static (GOT, PLT)
* Only part of the address changes
* Order of sections is (usually) preserved
* Can be leaked via /proc/self/maps
* Preserved between threads
* Controlled through /proc/sys/kernel/randomize\_va\_space

Position independent executable

* Program compiled to only use relative jump/ calls
* Doesn’t use static, hardcoded addresses
* Every section can be located anywhere from memory
* Sections are in the same position in relation to each other or are

Static vs dynamic

Static

* All libraries included in the binary
* Addresses known in advance as part of the executable
* No runtime resolving
* Huge size
* Lots of unnecessary code

Dynamic

* External functions loaded form external libraries
* Addresses resolved in runtime
* Smaller size
* Only a few functions exposed to direct addresses

Nx bit + Data Execution Prevention

* Disables on a CPU level ability to execute code from a given page in memory (rw-p)
* Disables execution from a stack and heap (no shellcode)
* Can be modified using mprotect syscall
* Usually only Read+Execute or Read+Write access to a page
* Page based for a CPU, not necessarily for a system

Stack canaries (protectors)

* Random number protecting return pointer
* Placed on a stack at the start of the function
* Checked just before returning
* Might be low entropy (predictability)
* Doesn’t protect other variables on the stack
* Can be bypassed if memory lead is present or arbitrary write

Shadow stack

* Return pointer is duplicated and placed on two stacks
* On return they are both compared
* If different then crash