<u>Threat Model</u> includes **assets** (what I am protecting/what matters most), **system goals** (functionality, security), **adversary definition** (risk assessment, includes goal + capability)

- **Goal**: what constitutes success? May involve subgoals (e.g. learn my 330 grade)
- **Capabilities**: what resources can adversary use (e.g., physical/remote access, access to source code, number of computers). Doesn't include strategy (can change)
  - Ex. View/send data to website, modify local browser state, access normal user account on AFS, invoke system calls on linux.andrew.cmu.edu

<u>Security properties</u>: **confidentiality/secrecy** (concealing information), **integrity** (prevent unauthorized changes), **authenticity** (data/actions attributed to correct person, no impersonation), **availability** (able to use resources when needed)

<u>Trusted Computing Base</u>: component X's TCB is all other components that must operate securely for X to be secure. **Trusted**: the system depends on component X. If X goes wrong/is subverted, so is the system. **Trustworthy**: there is reason to believe that your trust will not be misplaced. Code is type safe/formally verified. Code has been vetted by experts. Code is widely used and almost never has bugs. **<u>Trusted != Trustworthy</u>**

- Ideal TCB is verifiable (as small as possible) + tamper proof (don't mess with SSHD or OS executables, which can grant unauthorized users root access, compromise entire system)

<u>Key principles</u> of designing secure systems:

- <u>Economy of mechanism</u> (KISS): don't use a hammer when a gentle shove is enough
- <u>Fail-safe defaults</u>: most users don't change default settings, make defaults most secure
  - E.g. UNIX based on Access Control Lists deny access entirely if subject not listed on ACL
- <u>Don't rely on security by obscurity</u>: don't use your own functions, it will get leaked
- <u>Complete mediation</u>: every access to every object is checked by reference monitor
  - E.g. users have no direct access to disk (ask OS to provide access), can't snoop on others
- <u>Least privilege</u>: want users to have just enough powers to do what they need to do
- <u>Separation of duty</u>: no single person has enough power/need multiple to agree to launch
- <u>Defense in depth</u>: multiple walls; plan for defenses to fail, even if one does, exist more

<u>Program binary</u> (executable) segments: code (.TEXT), constants/globals (read-only .DATA)

- *objdump -d <exe>* for disassembly, *readelf -S <file>* for header/section/segment info
- **Little Endian**: addr m points at LSB, m+1 (higher address) goes into next byte
- **Stack alignment**: 16-byte aligned before call; before (*push %rbp*), rsp % 16 = 8

caller saves: rax, rdx, rcx, rsi, rdi, r8-r11	call pushes return addr to stack
arguments in rdi, rsi, rdx, rcx, r8, r9, and then stack	
callee saves: rbx, rbp, r12-r15	rbp often used to deref local vars
return value	pass back using rax
argument cleanup	caller's responsibility

<u>Format strings</u>: hhn to write as uint8_t, hn to write as uint16_t

- %9$s prints value of 9th argument on stack
- %9$n writes the number of bytes printed so far to the address stored at the 9th argument
  - %10u for 10 bytes of padding; %n writes to address of int passed as argument

Situation	**Data channel**	**Control Channel**	**Security**
Stack	Stack data	Return address	Control hijack
Format strings	Arguments	Format params	Read or write to memory
malloc buffers	malloc data	Heap metadata	Control hijack/write to memory

Mitigation	Attacks
ASLR	• nop (0x90) slides • Non-randomized memory (ret2text, func ptr): .TEXT not randomized • Stack juggling (ret2ret, ret2pop): bytecode from libc, etc • GOT hijacking (ret2got): initial GOT entry for printf points to code in linker, which updates address to transfer control to PLT entry of printf ○ *p system* to get address of *system@plt* (0x4010a0) ○ write 0xa0 at address of GOT entry, 0x10 at addr+1, etc.
Canaries	• Terminator canary (\x00) won't terminate memcpy() • Data pointer subterfuge: overwrite ptr (to return address) below canary
DEP/NX	• Return-to-libc: overwrite return address with system/execv, put arguments "/bin/sh" in correct registers

<u>CFI adversary model</u>: CAN overwrite any data memory at any time (stack, heap, data segments), CANNOT execute data, modify code, write to %ip or overwrite registers
Method: build CFG statically (at compile); rewrite binary (at install) to add checks before each control transfer; verify CFI instrumentation (load time); perform checks (run time)
Basic block: jump targets except at the beginning, no jumps except at the end ("straight")
Sound: analysis says X is true, then X is true | **Complete**: X is true, then analysis says its true
Analysis-precision tradeoffs: *<u>context</u>* (state called from; analyze procedure in isolation or based on each call site), *<u>flow</u>* (order of statements; variable type or set of variables modified by a procedure), *<u>path</u>* (whether conditionals are taken into account) sensitivity
Direct call: destination of call can't be changed without changing program text
Dynamic destinations (pointed at by indirect calls) are labelled equivalently if CFG contains edges to each from the same source.
<u>Safe</u>: program has not crashed and does not take undefined actions (exceptions are safe!)
<u>Type-safe</u> (language): well-typed programs always remain safe

<u>Hoare triple</u>: {Pre} Program {Post} s.t. if we start executing Program when *Pre* is true, it will terminate and *Post* will be true; **weakest precondition** is the most useful/broad

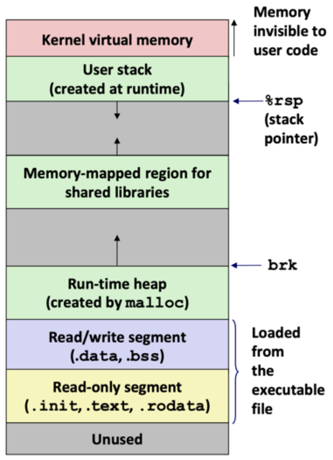
```
method concat(a:array<int>, b:array<int>)
returns (c:array<int>)
  ensures a[..] + b[..] == c[..]          {
  c := new int[a.Length + b.Length];
  var i := 0;
  while i < a.Length
    invariant 0 <= i <= a.Length
    invariant a[..i] == c[..i]            {
    c[i] := a[i];
    i := i + 1;                           }
  while i < c.Length
    invariant a.Length <= i <= c.Length
    invariant a[..] == c[..a.Length]
    invariant forall j :: |a| <= j < i
       ==> b[j - |a|] == c[j]             {
    c[i] := b[i - a.Length];
    i := i + 1;
  }
}
```

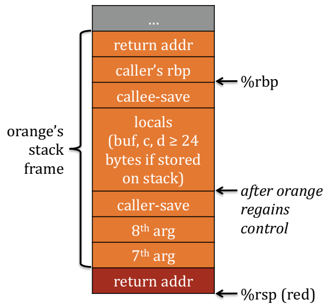
```
method double(input: seq<int>)
returns (output: seq<int>)
  ensures |output| == |input|
  ensures forall i :: 0 <= i < |input|
     ==> output[i] == 2 * input[i]
{
  output := input[..];
  var i := 0;
  while i < |output|
    invariant |input| == |output|
    invariant 0 <= i <= |output|
    invariant forall j :: 0 <= j < i
     ==> output[j] == 2 * input[j]
  {
    var val := input[i];
    output := output[i := 2 * val];
    i := i + 1;
  }
}
```

__Classes of program analysis__ (security analysis) to find code vulnerabilities:

- Human inspection: consider all semantic levels, humans are adaptive; code paths are deep and complex, must redo when code is changed
- Programmatic testing: use existing unit testing frameworks, hard to exercise every code path, number of test cases scale with code size, hard to identify root cause
- Randomized testing (fuzzing): randomly generate and permute inputs, monitor for undesired behavior (e.g. crash); simpler than programmatic testing, same cons
- Static analysis (FP, unknown vuln.): treat program as data/graph: can user-tainted data reach sensitive locations? Find bugs that violate relatively simple system rules that can be captured via state machines, but not bugs that involve complex control flow (e.g., via exceptions) or span multiple procedures or functional correctness bugs
- Dynamic analysis: symbolic execution (symbolic states + path constraints)
- Model checking: abstract away from actual code, prove property holds given program

Software security architecture: separation implemented with hard/software, VMs
Reference monitors: mediate crossing the confinement boundary
Software Fault Isolation: confine faults inside distrusted extensions but allow for efficient cross-domain calls. (1) **segment matching** checks every memory access for matching segment ID, slows down "common" intra-domain control transfer to speed up inter-domain transfer; (2) **sandboxing** force top bits to match segment ID and continue, no comparisons made, just ensures memory access stays in region (crash is ok)


Kernel virtual memory
Memory invisible to user code
User stack (created at runtime)
%rsp (stack pointer)
Memory-mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
Unused


...
return addr
caller's rbp
%rbp
callee-save
locals (buf, c, d ≥ 24 bytes if stored on stack)
orange's stack frame
caller-save
after orange regains control
8th arg
7th arg
return addr
%rsp (red)