Software Security: Vulnerability Analysis

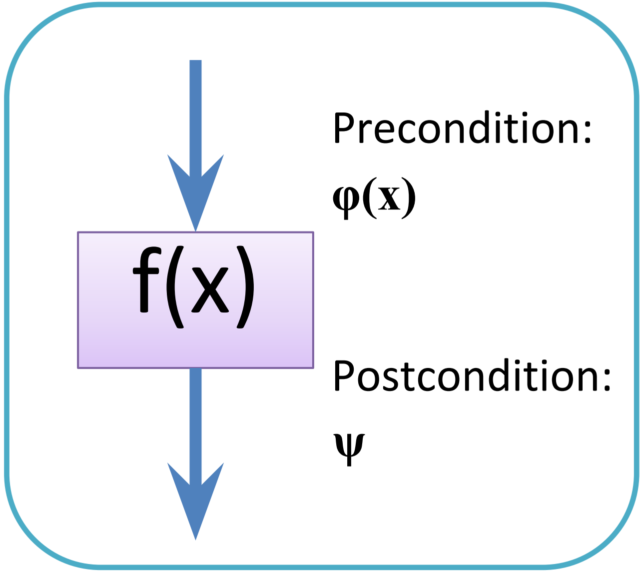
Program Verification

Program Verification

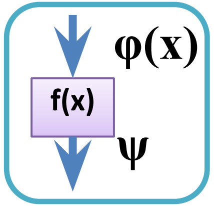
- How to prove a program free of buffer overflows?
  - Precondition
  - Postcondition
  - Loop invariants

Precondition

- Precondition for `f()` is an assertion (a logical proposition) that must hold at input to `f()`
  - If any precondition is not met, f() may not behave correctly
  - Callee may freely assume obligation has been met
- The concept similarly holds for any statement or block of statements



Precondition Example

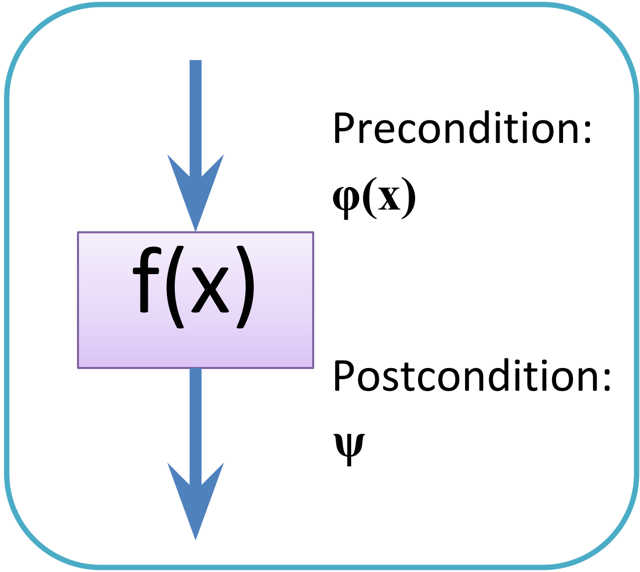


- Precondition:
  - fp points to a valid location in memory
  - fp points to a file
  - the file that fp points to contains at least 4 characters

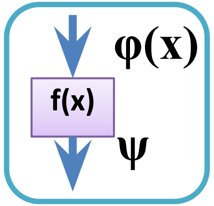
```
int parse(FILE *fp) {
  char cmd[256], *url, buf[5];
  fread(cmd, 1, 256, fp);
  int i, header_ok = 0;
  if (cmd[0] == ‘G’)
    if (cmd[1] == ‘E’)
      if (cmd[2] == ‘T’)
        if (cmd[3] == ‘ ’)
          header_ok = 1;
  if (!header_ok) return -1;
  url = cmd + 4;
  i=0;
  while (i<5 && url[i]!=‘\0’ && url[i]!=‘\n’) {
    buf[i] = tolower(url[i]);
    i++;
  }
  buf[i] = ‘\0’;
  printf(“Location is %s\n”, buf);
  return 0; }
```

Postcondition

- *Postcondition* for `f()`
  - An assertion that holds when `f()` returns
  - `f()` has obligation of ensuring condition is true when it returns
  - Caller may assume postcondition has been established by `f()`



Postcondition Example

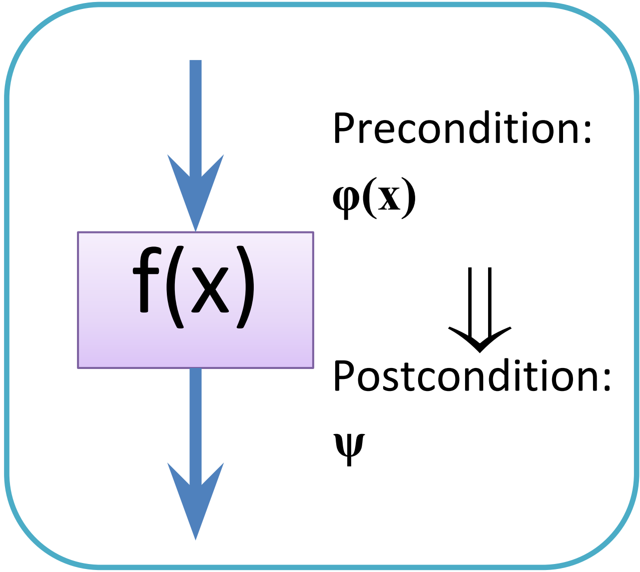


- Postcondition:
  - *buf* contains no uppercase letters
  - (return 0) $\Rightarrow (\mathrm{cmd}[0..3] == \text{“GET “})$

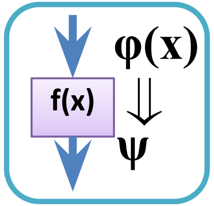
```
int parse(FILE *fp) {
  char cmd[256], *url, buf[5];
  fread(cmd, 1, 256, fp);
  int i, header_ok = 0;
  if (cmd[0] == ‘G’)
    if (cmd[1] == ‘E’)
      if (cmd[2] == ‘T’)
        if (cmd[3] == ‘ ’)
          header_ok = 1;
  if (!header_ok) return -1;
  url = cmd + 4;
  i=0;
  while (i<5 && url[i]!=‘\0’ && url[i]!=‘n’) {
    buf[i] = tolower(url[i]);
    i++;
  }
  buf[i] = ‘\0’;
  printf(“Location is %s\n”, buf);
  return 0; }
```

Proving Precondition $\Rightarrow$ Postcondition

- Given preconditions and postconditions
  - Specifying what obligations caller has and what caller is entitled to rely upon
- Verify: No matter how function is called,
  - if precondition is met at function's entrance,
  - then postcondition is guaranteed to hold upon function's return

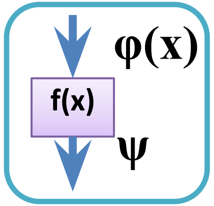


Proving Precondition $\Rightarrow$ Postcondition

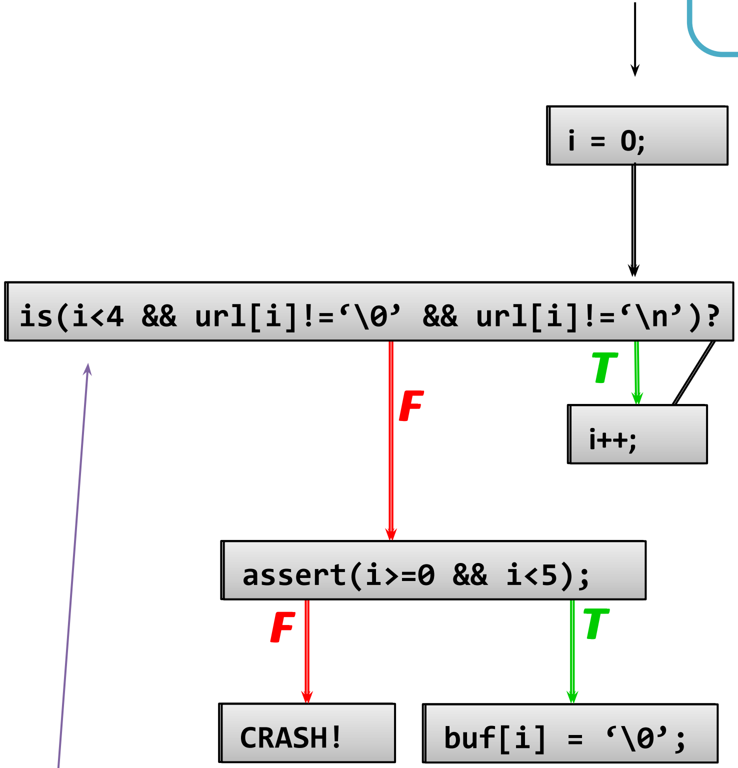


- Basic idea:
  - Write down a precondition and postcondition for every line of code
  - Use logical reasoning
- Requirement:
  - Each statement's postcondition must match (imply) precondition of any following statement
  - At every point between two statements, write down *invariant* that must be true at that point
    - Invariant is postcondition for preceding statement, and precondition for next one

We'll take our running example, fix the bug, and show that we can successfully prove that the bug no longer exists.

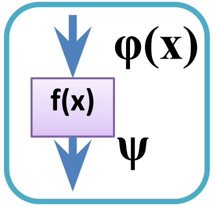



```
int parse(FILE *fp) {
  char cmd[256], *url, buf[5];
  fread(cmd, 1, 256, fp);
  int i, header_ok = 0;
  if (cmd[0] == 'G')
    if (cmd[1] == 'E')
      if (cmd[2] == 'T')
        if (cmd[3] == ' ')
          header_ok = 1;
  if (!header_ok) return -1;
  url = cmd + 4;
  i=0;
  while (i<4 && url[i]!='\0' && url[i]!='n') {
    buf[i] = tolower(url[i]);
    i++;
  }
  assert(i>=0 && i <5);
  buf[i] = '\0';
  printf("Location is %s\n", buf);
  return 0; }
```



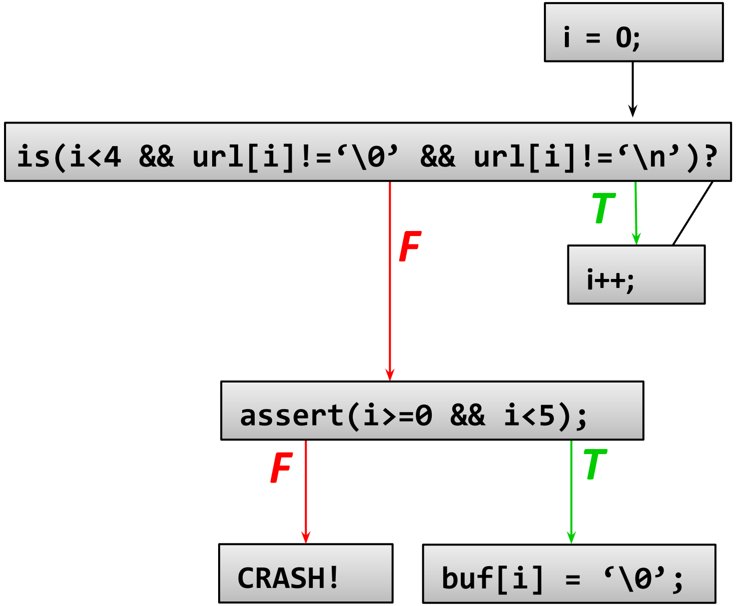
Bug Fixed!

We'll take our running example, fix the bug, and show that we can successfully prove that the bug no longer exists…



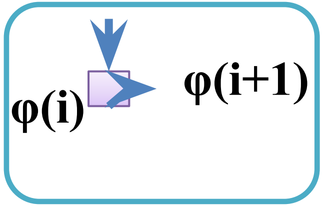

```
int parse(FILE *fp) {
  char cmd[256], *url, buf[5];
  fread(cmd, 1, 256, fp);
  int i, header_ok = 0;
  if (cmd[0] == 'G')
    if (cmd[1] == 'E')
      if (cmd[2] == 'T')
        if (cmd[3] == ' ')
          header_ok = 1;
  if (!header_ok) return -1;
  url = cmd + 4;
  i=0;
  while (i<4 && url[i]!='\0' && url[i]!='n') {
    buf[i] = tolower(url[i]);
    i++;
  }
  buf[i] = '\0';
  printf("Location is %s\n", buf);
  return 0; }
```

…So assuming fp points to a file that begins with "GET ", we want to show that *parse* never goes down the false assertion path.

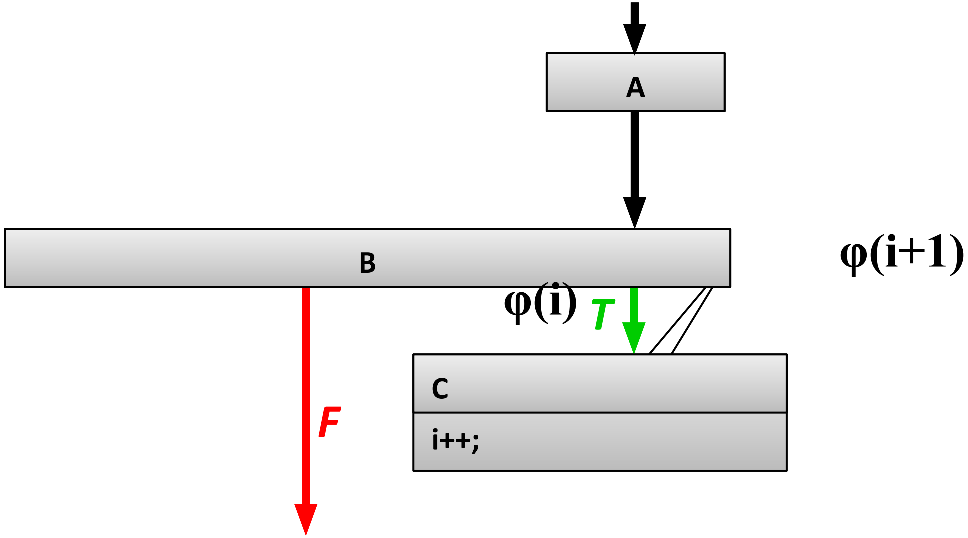


…But first, we will need the concept of loop invariant.

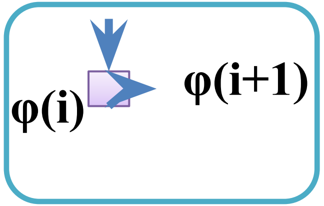
Loop Invariant and Induction



- An assertion that is true at entrance to the loop, on any path through the code
  - Must be true before every loop iteration
    - Both a pre- and post-condition for the loop body

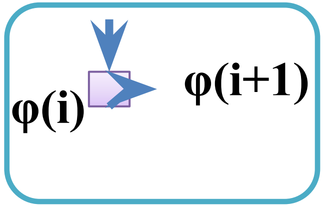


Loop Invariant and Induction

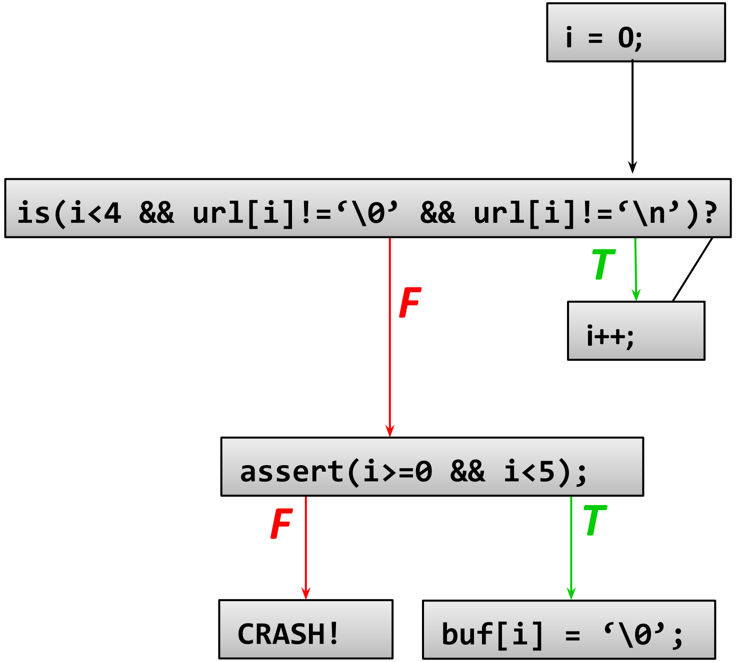


- To verify:
  - Base Case: Prove true for first iteration: **φ(0)**
  - Inductive step: Assume **φ(i)** at the beginning of the loop. Prove **φ(i+1)** at the start of the next iteration.

Try with our familiar example, proving that (0≤i<5) after the loop terminates:



LOOP INVARIANT: /* φ(i) = (0≤i<5) */



Base Case:

/* φ(0) = (0≤0<5) */

Inductive Step:

/* spp (0≤i<5) at the beginning of the loop */

/* for (0≤i<4), clearly (0≤i+1<5) */

/* (i=5) is not a possible case since that would fail the looping predicate */

/* ⇒ (0≤i+1<5) at the end of the loop */

/* ⇒ *parse* never fails the assertion */

Function Post-/Pre-Conditions

- For every function call, we have to verify that its precondition will be met
  - Then we can conclude its postcondition holds and use this fact in our reasoning
- Annotating every function with pre- and post-conditions enables *modular reasoning*
  - Can verify function `f()` by looking at only its code and the annotations on every function `f()` calls
    - Can ignore code of all other functions and functions called transitively
  - Makes reasoning about `f()` an almost purely local activity

Documentation

- Pre-/post-conditions serve as useful documentation
  - To invoke Bob's code, Alice only has to look at pre- and post-conditions – she doesn't need to look at or understand his code
- Useful way to coordinate activity between multiple programmers:
  - Each module assigned to one programmer, and pre-/post-conditions are a contract between caller and callee
  - Alice and Bob can negotiate the interface (and responsibilities) between their code at design time

Avoiding Security Holes

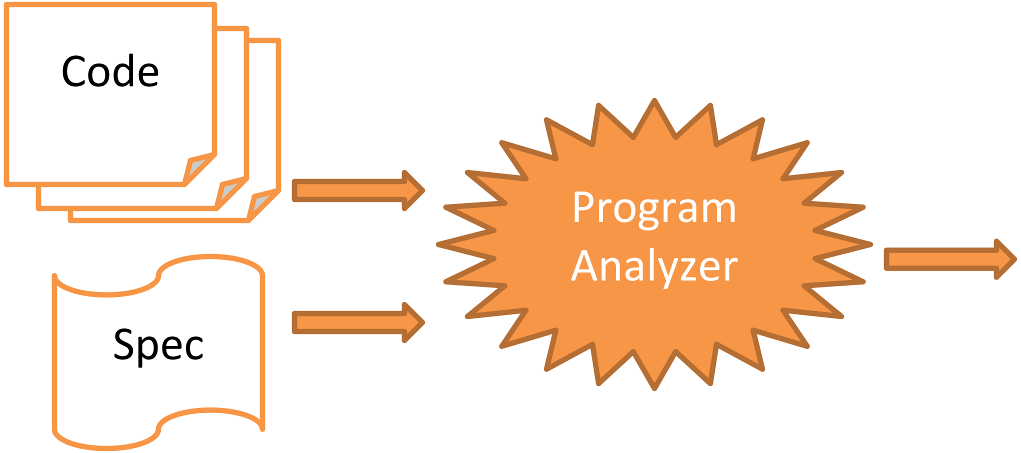
- To avoid security holes (or program crashes)
  - Some implicit requirements code must meet
    - Must not divide by zero, make out-of-bounds memory accesses, or deference null ptrs, …
- Prove that code meets these requirements using same style of reasoning
  - Ex: when a pointer is dereferenced, there is an implicit precondition that pointer is non-null and in-bounds

Avoiding Security Holes

- Proving absence of buffer overruns might be much more difficult
  - Depends on how code is structured
- Instead of structuring your code so that it is hard to provide a proof of no buffer overruns, restructure it to make absence of buffer overruns more evident

- Lots of research into automated theorem provers to try to mathematically prove validity of alleged pre-/post-conditions
  - Or to help infer such invariants

Program Analyzers



Report	Type	Line
1	stack oflow	324
2	buffer oflow	8,491
3	buffer oflow	23,212
4	mem leak	86,923
5	unsafe indexing op	5,393,245
…	…	…
12,002	info leak	10,921

false alarm (report 2)

false alarm (report 4)

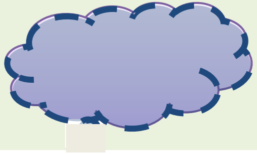
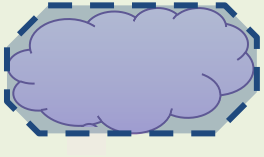
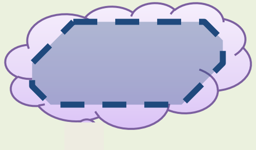
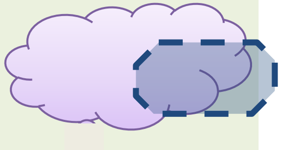
analyze large code bases

potentially reports many warnings

may emit false alarms

Soundness, Completeness

Property	Definition
Soundness	If the program contains an error, the analysis will report a warning. “Sound for reporting correctness”
Completeness	If the analysis reports an error, the program will contain an error. “Complete for reporting correctness”

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable** *(Ex: Manual Program Verification)* 	Reports all errors May report false alarms **Decidable** *(Ex: Abstract Interpretation)* 
Unsound	May not report all errors Reports no false alarms **Decidable** *(Ex: Symbolic Execution)* 	May not report all errors May report false alarms **Decidable** *(Ex: Syntactic Analysis)* 

Isolation and Reference Monitor

Slide credit: Dan Boneh

Running untrusted code

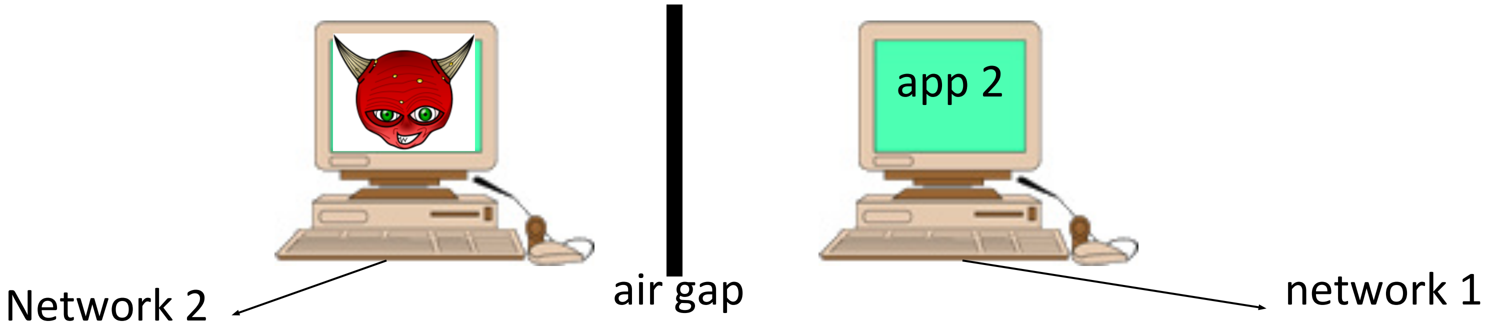
We often need to run buggy/untrusted code:

- programs from untrusted Internet sites:
    - toolbars, viewers, codecs for media player
- old or insecure applications: ghostview, outlook
- legacy daemons: sendmail, bind
- Honeypots

- Goal: ensure misbehaving app cannot harm rest of system
- Approach: Confinement
    - Can be implemented at many different levels

Confinement (I): Hardware

- **Hardware:** run application on isolated hw (air gap)

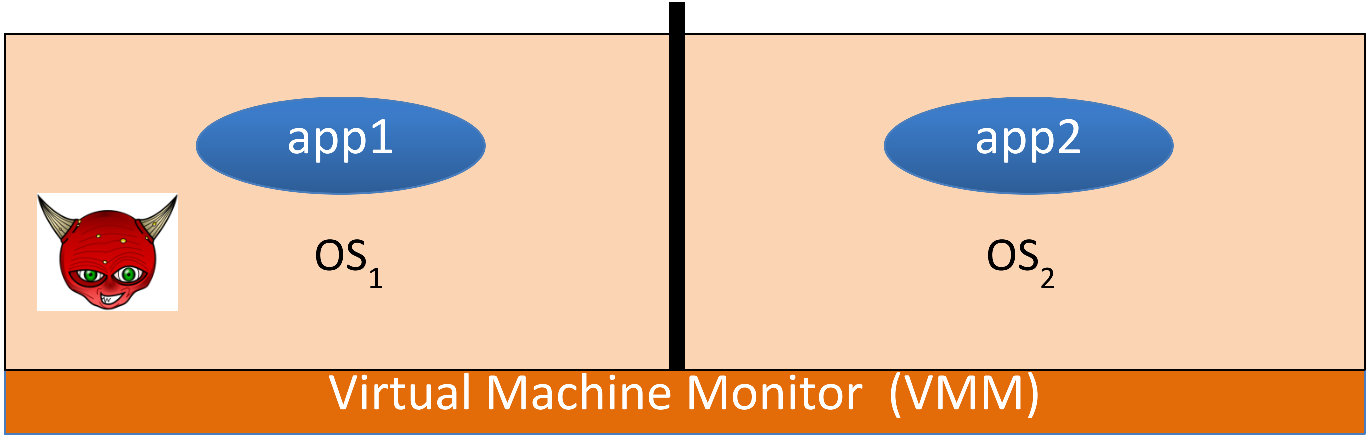


Confinement (II): Firewall

- Firewall: isolate internal network from the Internet

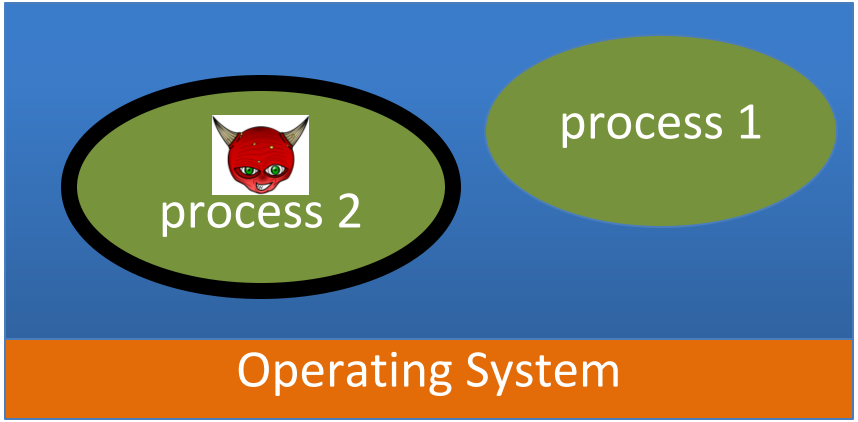
Confinement (III): VM

- **Virtual machines**: isolate OS's on a single machine



Confinement (IV): Processes

- **Processes:**
  - Isolate a process in a single operating system
  - System Call Interposition



Confinement (V): SFI

- **Threads:** Software Fault Isolation (SFI)
    - Isolating threads sharing same address space

Implementing confinement: Reference Monitor

Key properties:

- **Mediates requests** from applications
  - Implements protection policy
  - Enforces isolation and confinement
- Must **<u>always</u>** be invoked (complete mediation)
  - Every application request must be mediated
- **Tamperproof**/fail safe
  - Reference monitor cannot be killed
  - or if killed, then monitored process cannot accessing anything requiring reference monitor’s approval
- **Small** enough to be analyzed and validated