Software Fault Isolation

Chester Rebeiro

Indian Institute of Technology Madras

Web Browser Confinement

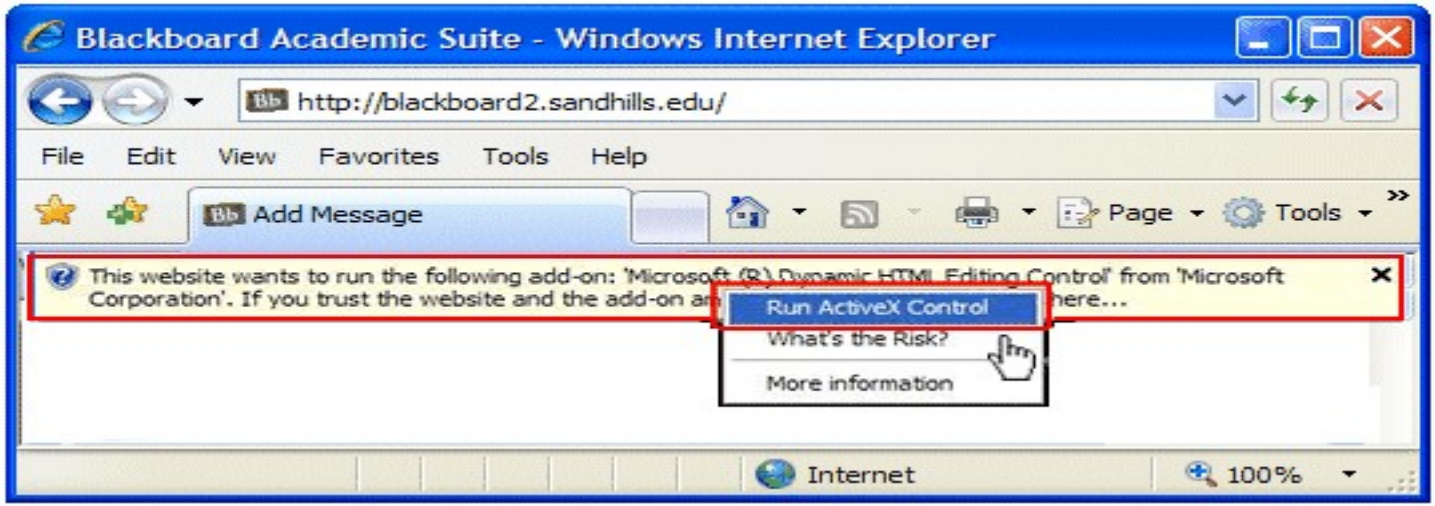
- Why run C/C++ code in web browser
  - Javascript highly restrictive / very slow
  - Not suitable for high end graphics / web games
  - Would permit extensive client side computation

- Why not to run C/C++ code in web browser
  - Security!
    Difficult to trust C/C++ code

Web Browser Confinement

- How to allow an untrusted module to load into a web-browser?
  - Trust the developer / User decides

    Active X



Web Browser Confinement

- How to allow an C/C++ in a web-browser?
  - Trust the developer / User decides

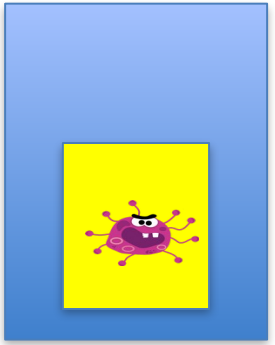
    Active X
  - Fine grained confinement
    - (eg. NACL from Google)
    - Uses Software Fault Isolation

Fine Confinement within a Process

- How to
  - restrict a module's capabilities
  - Restrict read/modification of data in another module

(jumping outside a module and access data outside a module should be done only through prescribed interfaces)

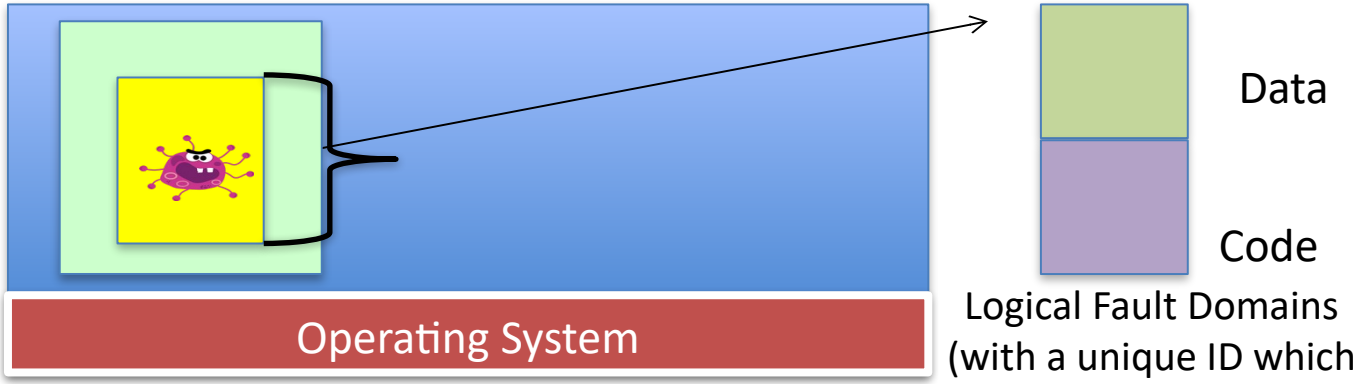
(can use RPCs, but huge performance overheads)



Application

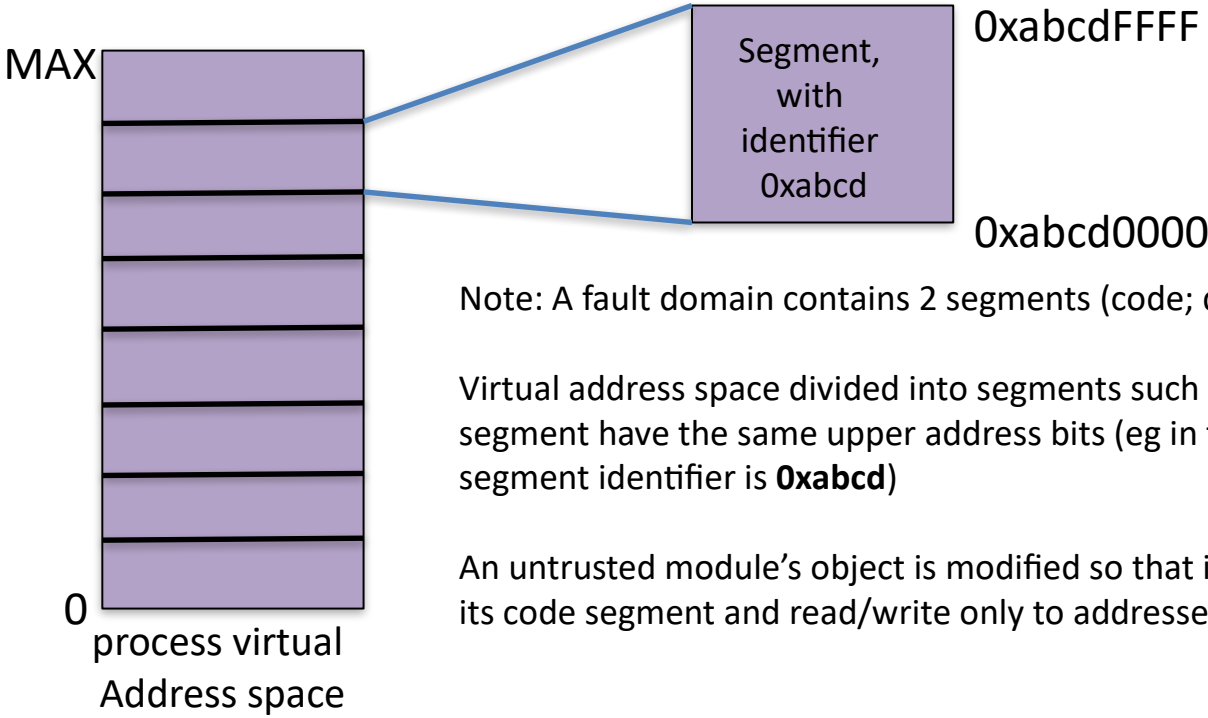
Fine Grained Confinement
(Software Fault Isolation)

- process space partitioned into logical domains.
- Each fault domain contains data, code, and a unique ID
- Code in one domain not allowed to read/modify data in another domain.
- Code in one domain cannot jump to another domain.
- The only way is through a low cost cross-fault-domain RPC interface not involving the OS..



Wahbe et al. Efficient Software Fault Isolation, SOSP 93
https://dl.acm.org/citation.cfm?doid=168619.168635

Segments and Segment Identifier

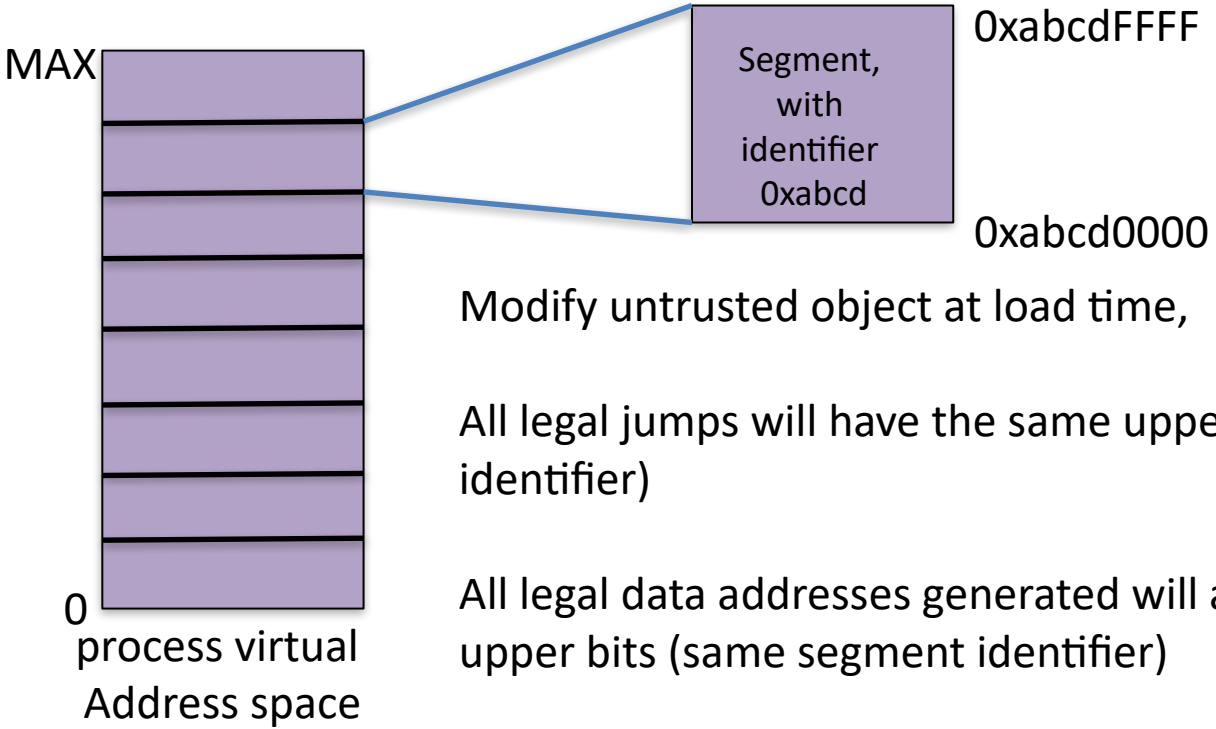


Note: A fault domain contains 2 segments (code; data+stack+heap)

Virtual address space divided into segments such that addresses in the same segment have the same upper address bits (eg in the above example the segment identifier is **0xabcd**)

An untrusted module's object is modified so that it can jump only to targets in its code segment and read/write only to addresses within its data segment.

Segments and Segment Identifier



Modify untrusted object at load time,

All legal jumps will have the same upper bits (same segment identifier)

All legal data addresses generated will also have the same upper bits (same segment identifier)

Achieving Segmentation

- Binary rewriting statically
  - At the time of loading, parse through the untrusted module to determine all memory read and write instructions and jump instructions.
  - Use unique ID (upper bits) to determine if the target address is legal
  - Rewriting can be done either at compile time (modifying compiler) or at load time.
  - A verifier is also needed when the module is loaded into the fault domain.

Safe & Unsafe Instructions

Safe Instructions:

- Most instructions are safe (such as ALU instr)
- Many of the target addresses can be resolved statically (jumps and data addresses within the same segment id. These are also **safe instructions**)

Safe Instructions

- Compile time techniques / Load time techniques
  - Scan the binary from beginning to end.
  - Reliable disassembly: by scanning the executable linearly
    - variable length instructions may have issues

```
25  CD  80  00  00
AND %eax, 0x000080CD
```

```
CD  80  00  00
INT $0x80
```

    - A jump may land in the middle of an instruction
    - Two ways to deal with this—
      - Ensure that all instructions are at 32 byte offsets
      - Ensure that all Jumps are to 32 byte offset

```
AND eax, 0xffffffe0
JMP *eax
```

Unsafe Instructions

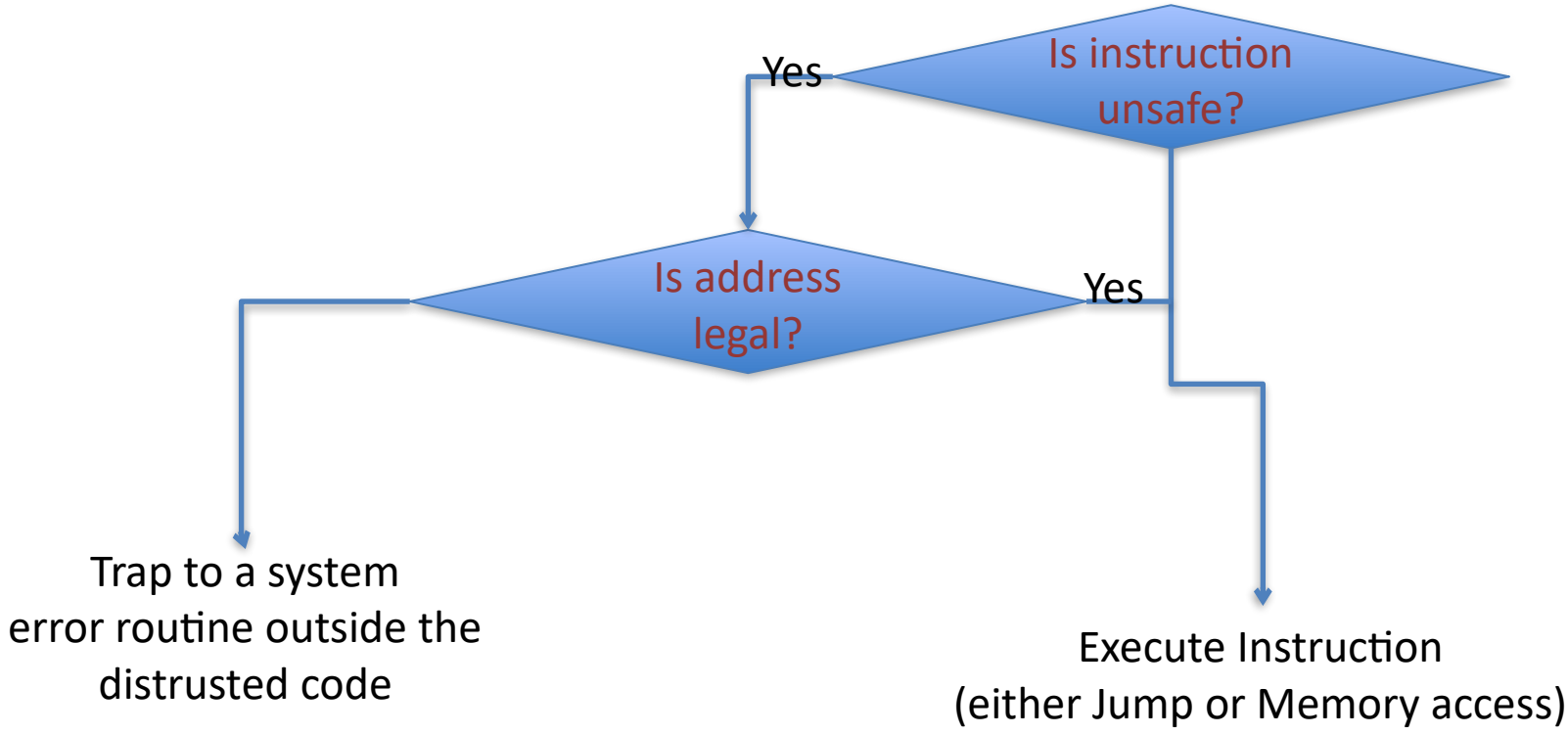
Prohibited Instructions:

- Eg. int, syscall, etc.

Unsafe Instructions: Cannot be resolved statically.

- For example *store 0x100, [r0]*
- Unsafe targets need to be validated at **runtime**
- Jumps based on registers (eg. Call *eax), and Load/stores that use indirect addressing are unsafe.
  Eg. JMP *eax

Runtime Checks for Unsafe Instructions (segment matching)



Run Time Checks Segment Matching

Insert code for every unsafe instruction that would trap if the store is made outside of the segment

4 registers required (underlined registers)

```
dedicated-reg ← target address
scratch-reg ← (dedicated-reg >> shift-reg)
compare scratch-reg and segment-reg
trap if not equal
store/jump using dedicated-reg
```

Overheads increase due to additional instructions but the increase is not as high as with RPCs across memory modules.

Address Sandboxing

- Segment matching is strong checking.
  - Able to detect the faulting instruction (via the trap)
- Address Sandboxing : Performance can be improved if this fault detection mechanism is dropped.
  - Performance improved by not making the comparison but forcing the upper bits of the target address to be equal to the segment ID
  - Cannot catch illegal addresses but prevents module from illegally accessing outside its fault domain.

Segment Matching : Check :: Address Sandboxing : Enforce

Address Sandboxing

Requires 5 dedicated registers

```
dedicated-reg^x2 ← target-reg & and-mask-reg
dedicated-reg ← dedicated-reg | segment-reg^x2
store/jump using dedicated-reg
```

Enforces that the upper bits of the dedicated-reg contains the segment identifier

Ensure Valid Instructions

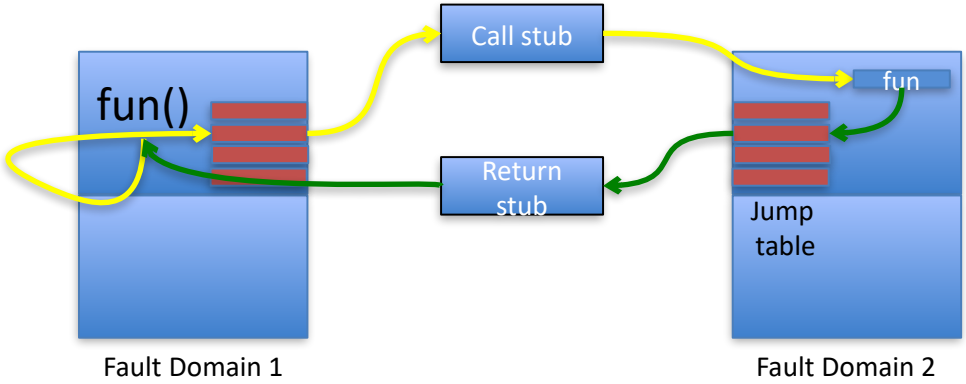
- How to ensure that jump targets are at valid instruction locations
  - Ensure that all instructions are at 32 byte offsets
  - Ensure that all Jumps are to 32 byte offset

```
AND eax, 0xffffffe0
JMP *eax
```

```
25  CD  80  00  00
AND %eax, 0x000080CD
```

```
CD  80  00  00
INT $0x80
```

Calls between Fault Domains
(light weight cross-fault-domain-RPC)



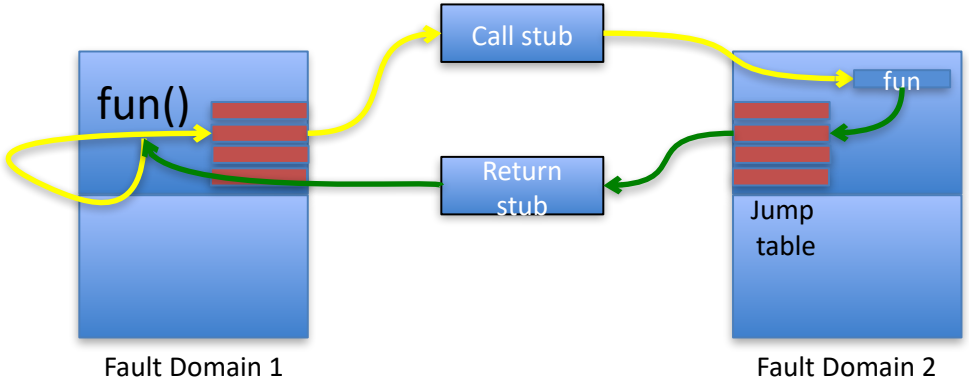
Safe calls outside a fault domain is by jump tables.

Each entry in jump table is a control transfer instruction whose target address is a legal entry point outside the domain.

Maintained in the read only segment of the program therefore cannot be modified.

Calls between Fault Domains (cross-fault-domain-RPC)

- A pair of stubs for each pair of fault domains
- Stubs are trusted
- Present outside the fault domains
- Responsible for
  - copying cross-domain arguments between domains
  - manages machine state (store/restore registers as required)
  - Switch execution stack
  - They can directly copy call arguments to the target domain



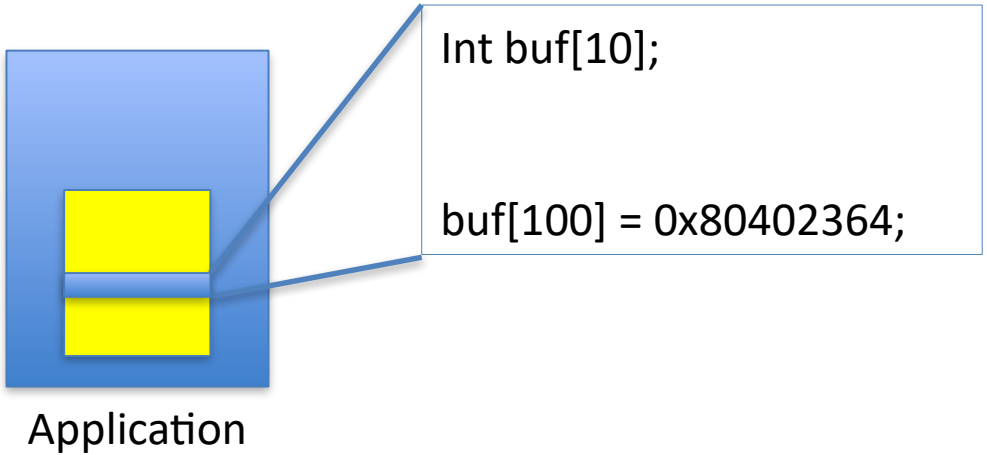
- Cheap
  - No traps, no context switches

System Resources

- How to ensure that one fault domain does not alter system resources used by another fault domain
  - For example, does not close the file opened by another domain
- One way,
  - Let the OS know about the fault domains
  - So, the OS keeps track if such violations are done at the system level
- Another (more portable way),
  - Modify the executable so that all system calls are made through a well defined interface called *cross-fault-domain-RPC*.
  - The cross-fault-domain-RPC will make the required checks.

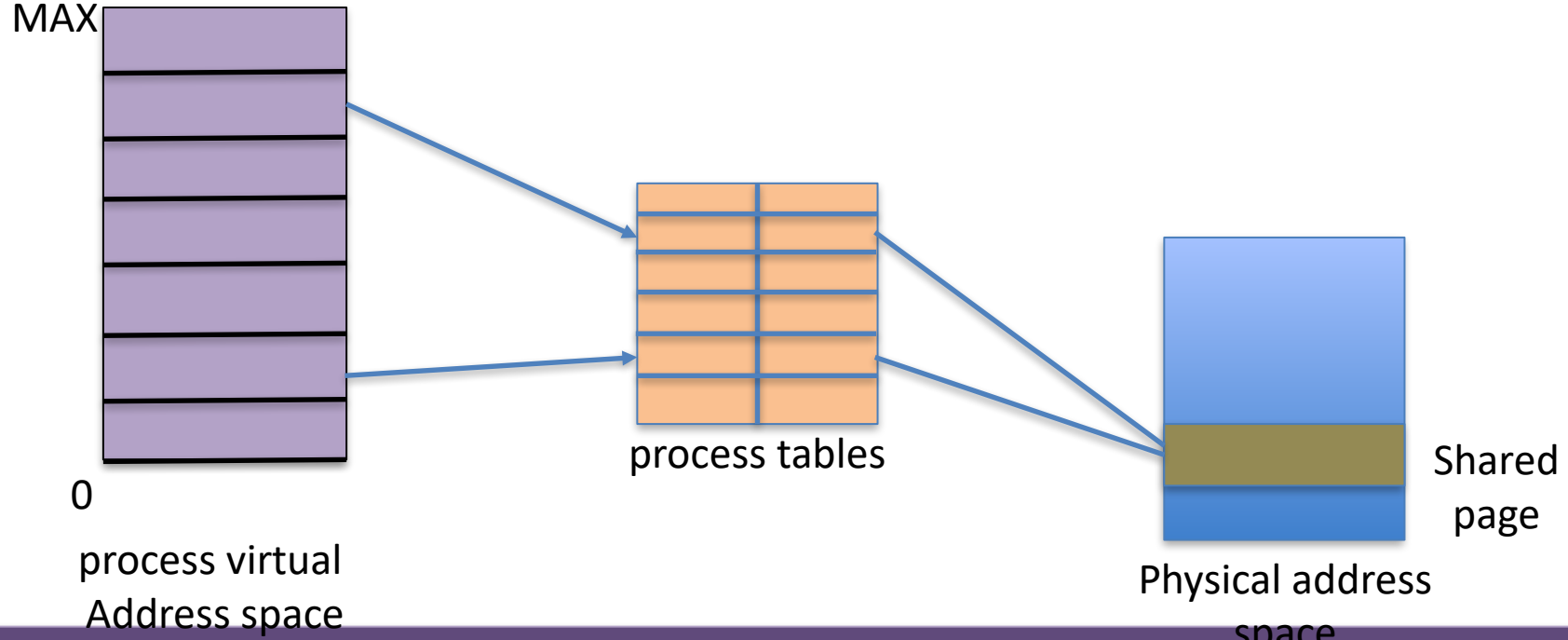
Points to Ponder

What happens if there is a buffer overflow in the isolated code?

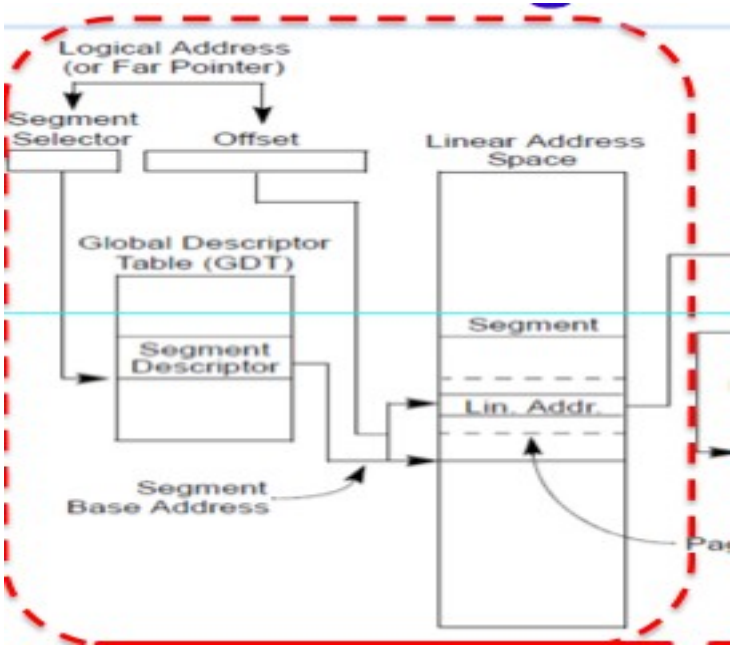


Shared Data (Global / Heap Variables)

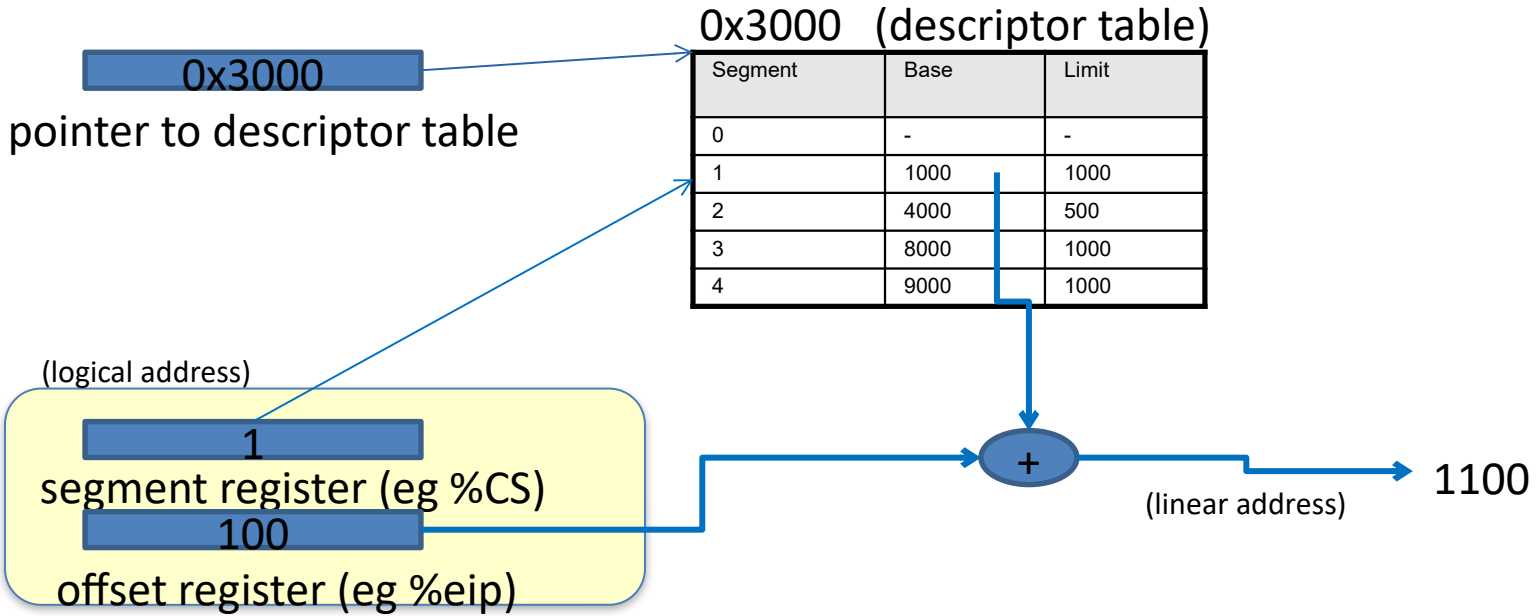
- Page tables in kernel modified so that shared memory mapped to every segment that needs access to it



Segmentation
(Hardware Support for Sandboxing)



Segmentation Example



Segmentation In Sandboxing

- Create segments for each sandbox
- Make segment registers (CS, ES, DS, SS) point to these segments

- Need to ensure that the untrusted code does not modify the segment registers

- Jumping out of a segment: need to change segment registers appropriately