CSE 127: Computer Security

Isolation and secure design

Deian Stefan

Today

Lecture objectives:

- Understand basic principles for building secure systems
- Understand mechanisms used to build secure systems

We often need to run untrusted code

- Desktop applications
- Mobile apps
- Websites, browser extensions, PDFs
- VMs running in the cloud
- Serverless functions

Systems must be designed to be resilient in the face of vulnerabilities and malicious users

Principles of secure design

- Least privilege
- Privilege separation
- Complete mediation
- Fail safe/closed
- Keep it simple
- Defense in depth

Least privilege

- Users should only have access to data and resources needed to do their task
- Example:

Least privilege

- Users should only have access to data and resources needed to do their task
- Example:
  - ➤ Faculty can only change grade for classes they teach
  - ➤ Only employees with background checks have access to classified documents
- How do you enforce least privilege?

Privilege separation

The most common way to enforce least privilege is to use privilege separation

- Break system into compartments
- Ensure each compartment is isolated
- Ensure each compartment runs with least privilege
- Treat compartment interface as trust boundary

Some photos from Smith's *A Contemporary Look at Saltzer and Schroeder's 1975 Design Principles and Wikipedia (e.g.,* https://en.wikipedia.org/wiki/Beaumaris_Castle)

Example: Multi-user OS

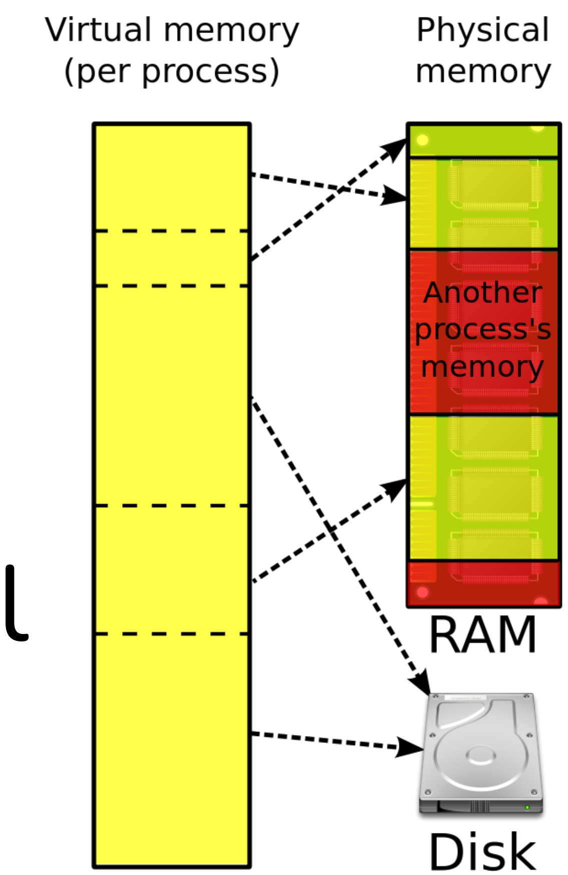
- In this system:
  - Users can execute programs (process)
  - Processes can access resources/assets
- What's the threat model?
- What are the assets
- What security properties do we want to preserve?

Multi-user OS properties

- Memory isolation
  - ➢ Process should not be able to access another’s memory
- Resource isolation
  - ➢ Process should only be able to access certain resources

Memory isolation

- How are individual processes memory-isolated from each other?
  - ➤ Each process gets its own virtual address space, managed by the operating system
- Memory addresses used by processes are virtual addresses (VAs) not physical addresses (PAs)
  - ➤ When and how do we do the translation?



https://en.wikipedia.org/wiki/Virtual_memory#/media/File:Virtual_memory.svg

Principle of complete mediation

- Every access goes through address translation
  - Load, store, instruction fetch
- Who does the translation?

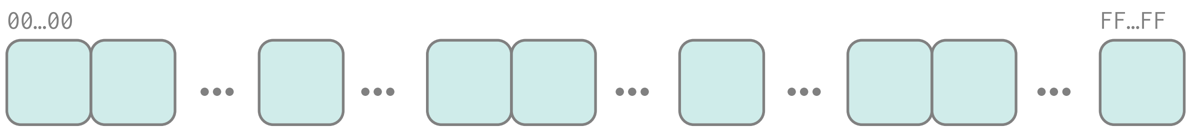
Principle of complete mediation

- Every access goes through address translation
  - Load, store, instruction fetch
- Who does the translation?
  - The CPU’s memory management unit (MMU)

How does the MMU translate VAs to PAs?

- How do we translate arbitrary 64bit addresses?
  - We can’t map at the individual address granularity!
  - 64 bits * 2^{64} (128 exabytes) to store any possible mapping

Address translation (closer)

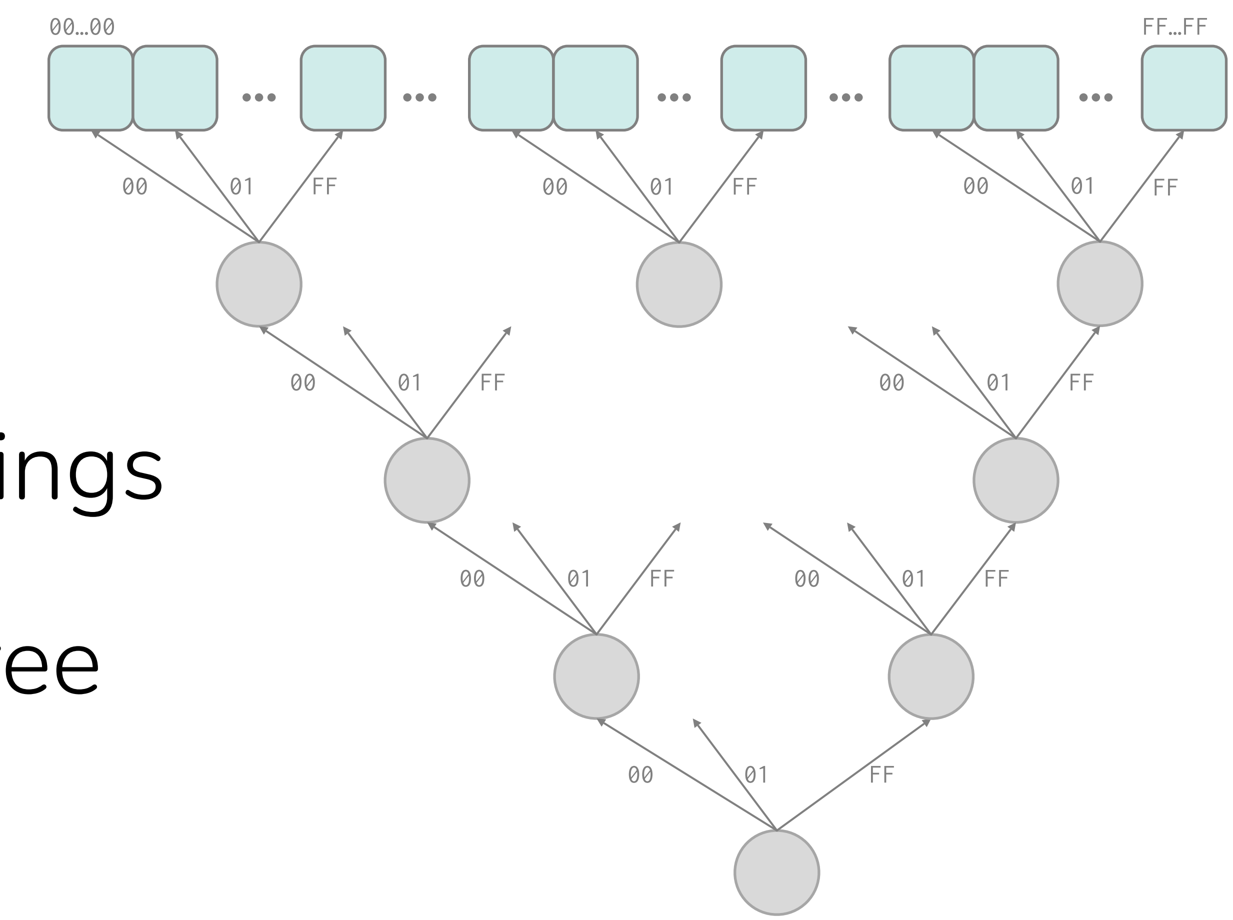


- Page: basic unit of translation
  - ➤ Usually 4KB = 2^{12}
- How many page mappings?
  - ➤ Still too big!
  - ➤ 52 bits * 2^{52} (208 petabytes)

So what do we actually do?

Multi-level page tables

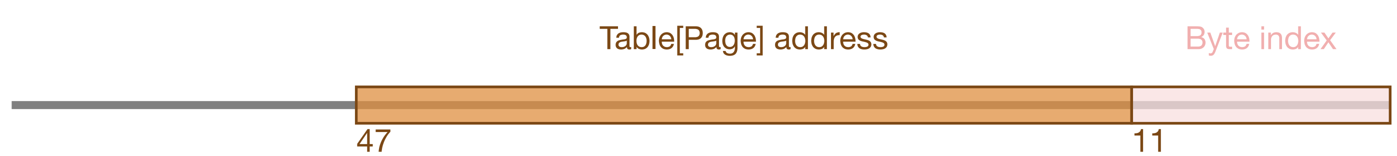
- Sparse tree of page mappings
- Use VA as path through tree
- Leaf nodes store PAs
- Root is kept in register so MMU can walk the tree



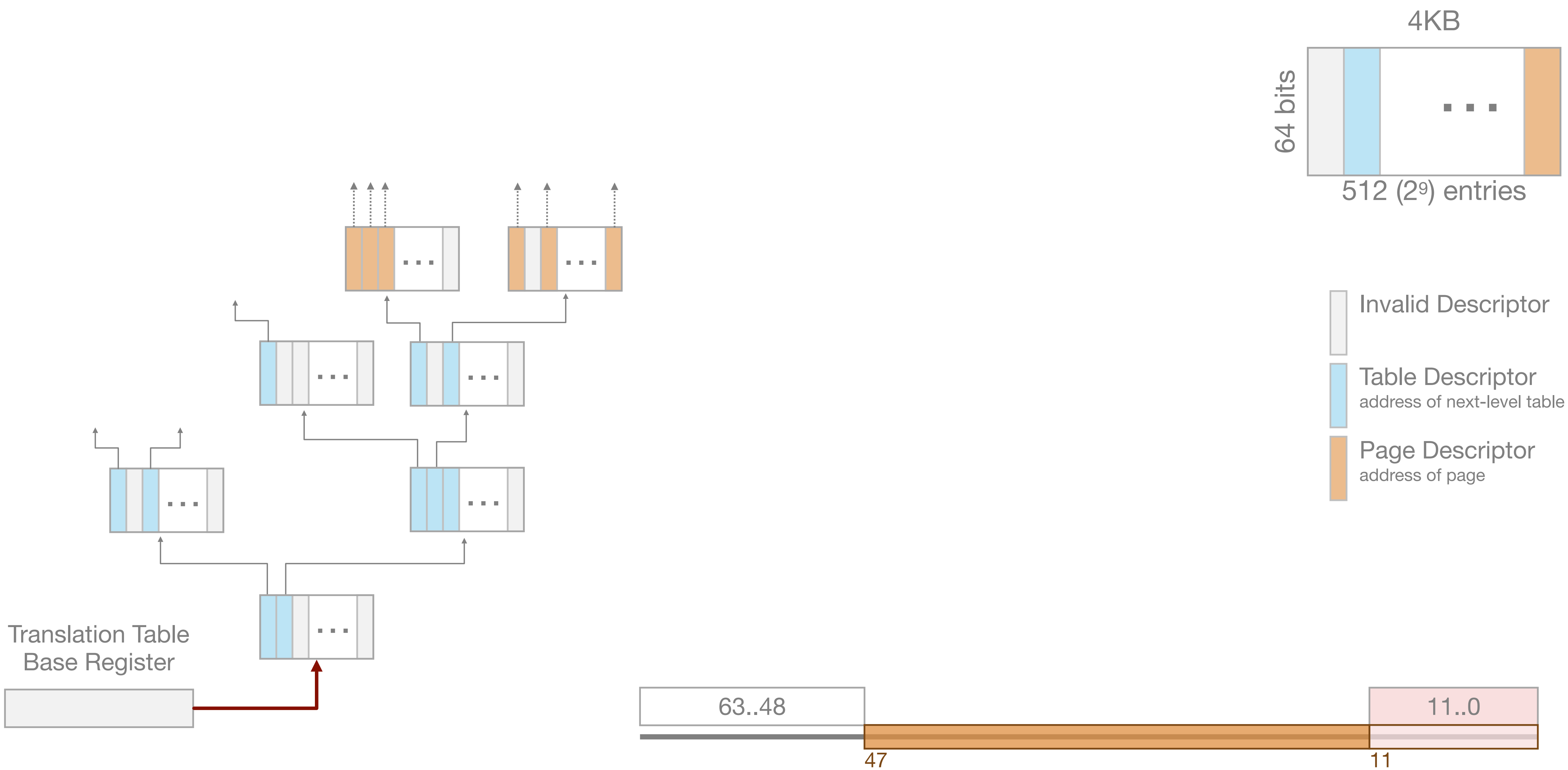
Example of page table walk

In reality, the full 64bit address space is not used.

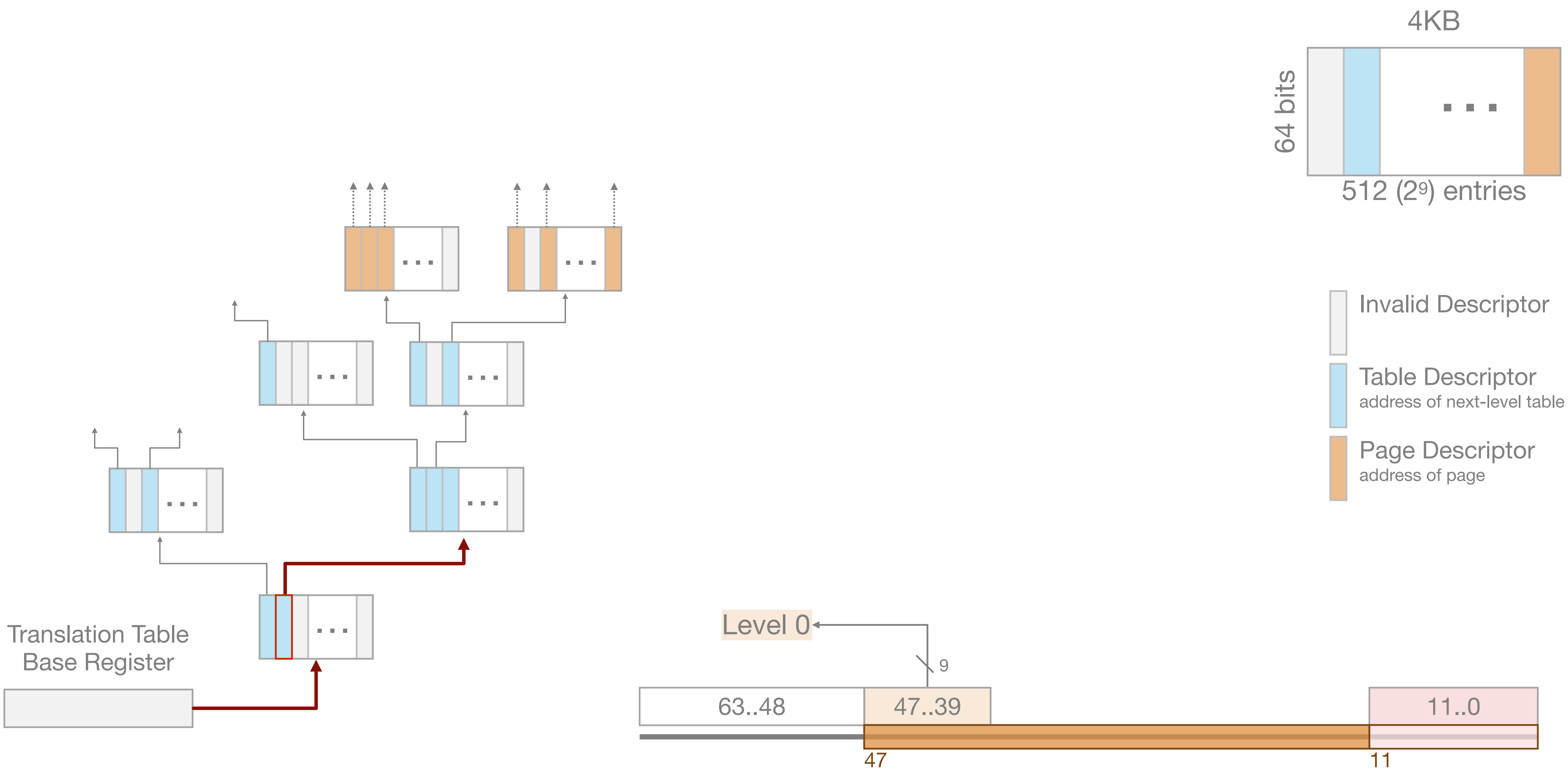
- Working assumption: 48bit addresses



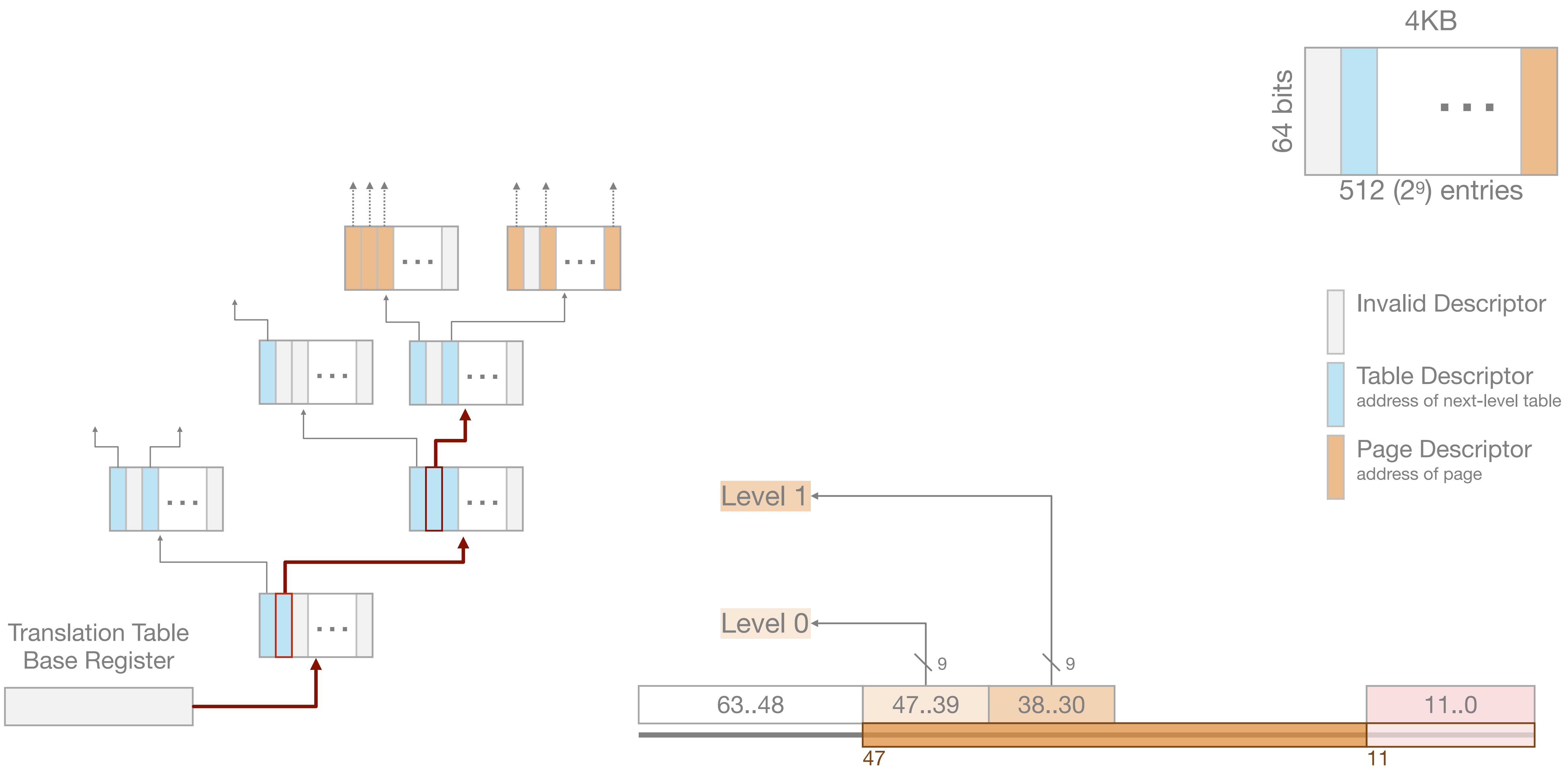
Page table walk



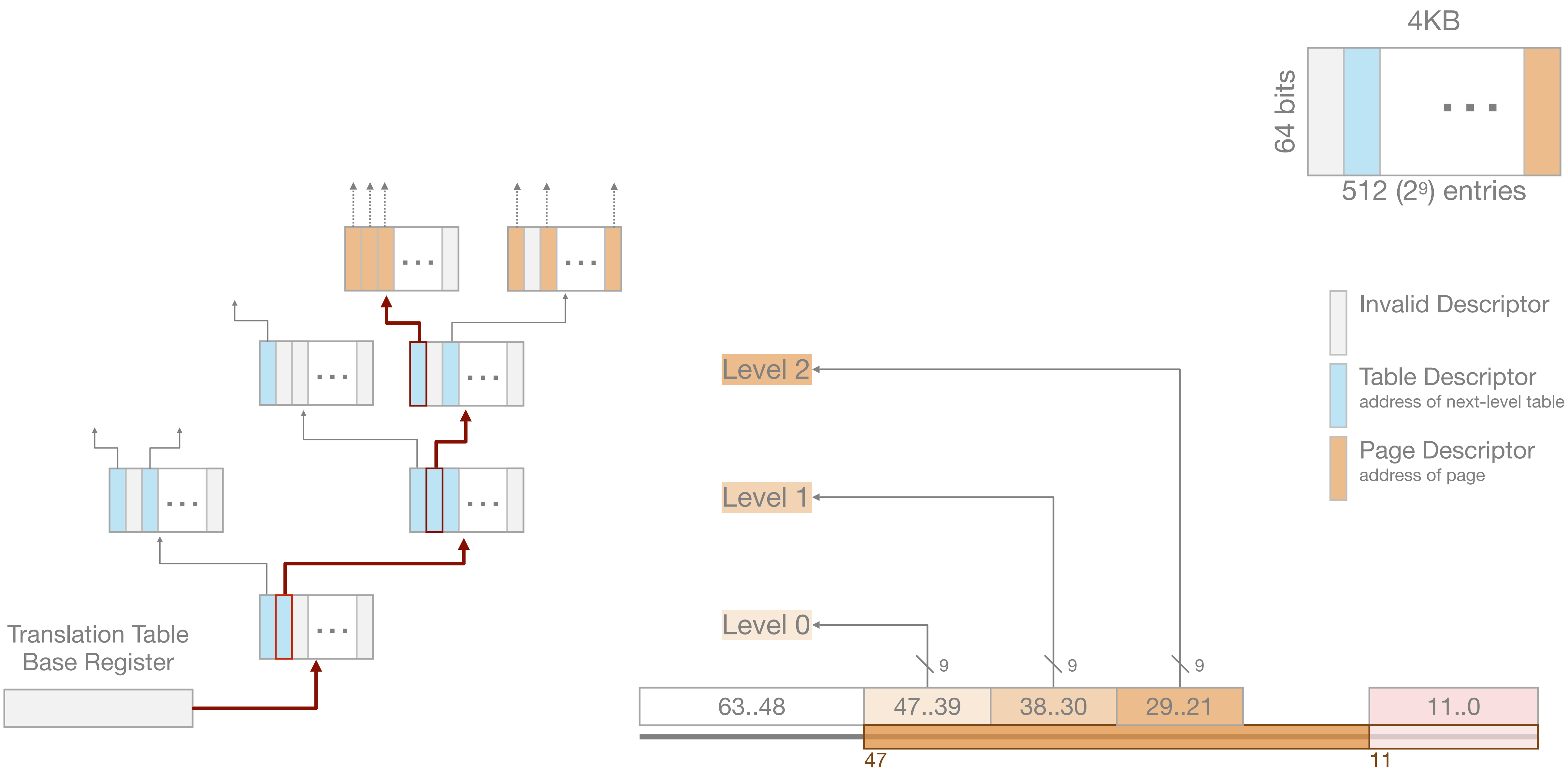
Page table walk



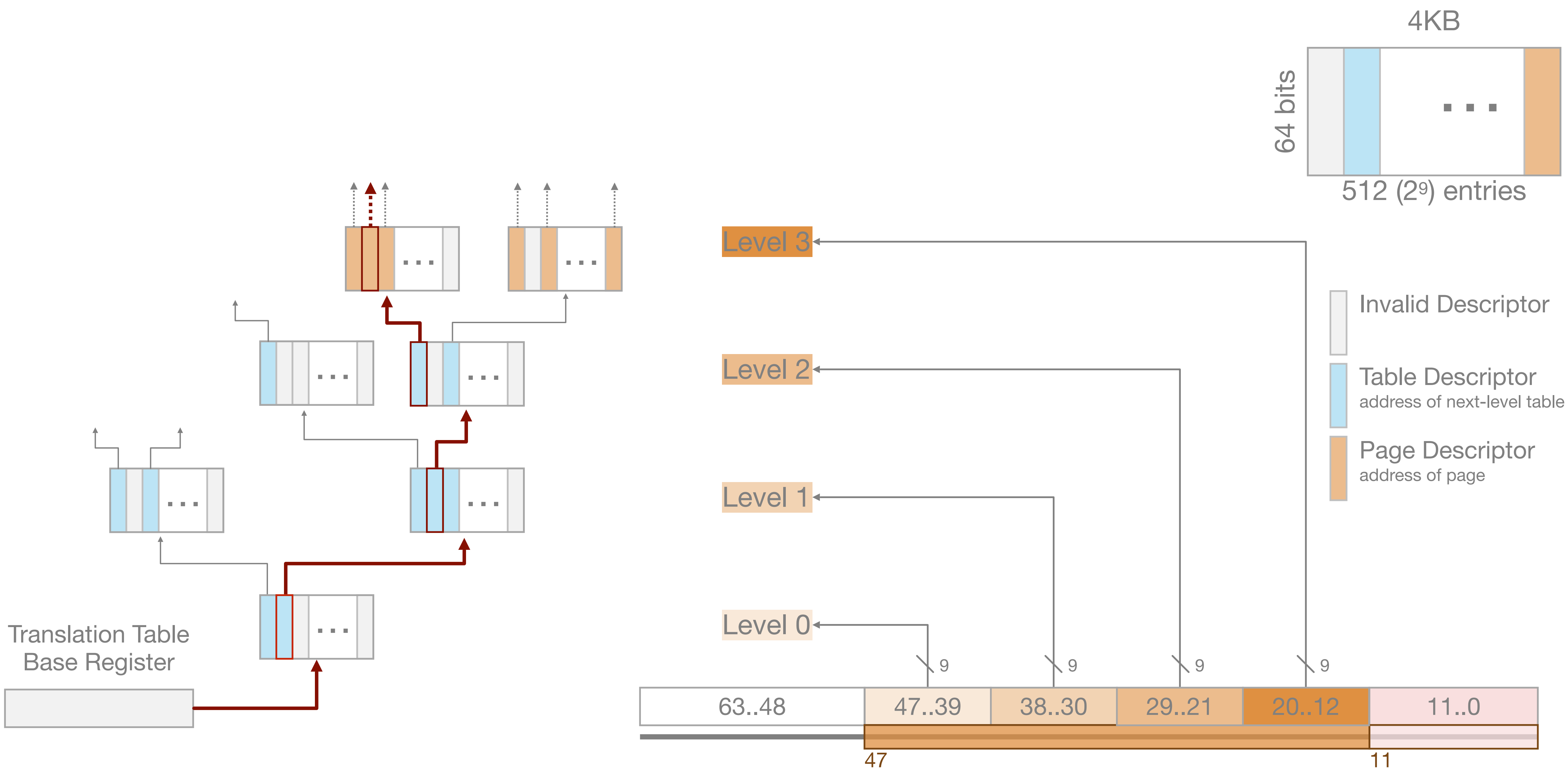
Page table walk



Page table walk



Page table walk



Do we actually perform all these memory looks on every load and store?

The translation lookaside buffer

The translation lookaside buffer

- Small cache of recently translated addresses
  - Before translating a referenced address, the processor checks the TLB
- What does the TLB give us?

The translation lookaside buffer

- Small cache of recently translated addresses
  - Before translating a referenced address, the processor checks the TLB
- What does the TLB give us?
  - Physical page corresponding to virtual page (or that page isn't present)

The translation lookaside buffer

- Small cache of recently translated addresses
  - Before translating a referenced address, the processor checks the TLB
- What does the TLB give us?
  - Physical page corresponding to virtual page (or that page isn't present)
  - Access control: if mapping allows the mode of access

How do we get isolation?

- Each process gets its own tree
  - ➢ Tree is created by the OS
  - ➢ Tree is used by the MMU when doing translation
    - ➢ This is called “page table walking”
- When you context switch:
  - ➢ OS changes the root
  - ➢ What about the TLB?

Multi-user OS properties

- Memory isolation
  - Process should not be able to access another's memory
- Resource isolation
  - Process should only be able to access certain resources

Resource isolation in UNIX

Everything is a file: files, sockets, devices, etc.

- Permissions to access files governed by user IDs
  - Every user has a unique UID
  - Each file has an access control list:
    how the (owner, group, others) can access the file
  - root (UID 0) can access everything

```
$  ls -l /fast
total 56
drwxrws---  4 root     adverse      4096 Feb 23  2018 adverse
drwxr-sr-x 12 cdisselk adverse      4096 Nov 16  2018 adverse-gipeda
drwxr-sr-x  3 root     adverse      4096 Jan  3  2019 ccfiles
-rwxr-xr-x  1 rsw      adverse      4343 Feb 20  2019 dclang
drwxr-sr-x  7 rsw      adverse      4096 Feb 20  2019 llfiles
drwx------  2 root     root        16384 Feb 14  2018 lost+found
drwxr-sr-x  7 mlfbrown mlfbrown     4096 Jan 20  2020 mlfbrown
drwxr-sr-x  3 cdisselk adverse      4096 Apr 17  2019 persisted
drwxrwsr-x  8 d        proofmonkey  4096 Nov 21  2019 proofmonkey-gecko-dev.git
drwxr-xr-x  2 root     root         4096 Aug 26  2020 vms
```

Process UIDs

- Real user ID (RUID)
  - ➤ Used to determine which user started the process
  - ➤ Typically same as the user ID of parent process
- Effective user ID (EUID)
  - ➤ Determines the permissions for process
  - ➤ Can be different from RUID (e.g., because `setuid` bit on the file being executed)
- Saved user ID (SUID)
  - ➤ EDUI prior to change

`setuid`

- A program can have a `setiud` bit set in its permissions

```
-rwsr-xr-x 1 root root 55440 Jul 28  2018 /usr/bin/passwd
```

- This impacts fork and exec
  - Typically inherit three IDs of parent
  - If `setuid` bit set: use UID of file owner as EUID
- Why do we need this?

`setgid` and sticky bit

- setgid on file: set $EG_{roup}ID$ of process to GID of file
  on directory: sets group of files within

```
drwxrws---  4 root     adverse       4096 Feb 23  2018 adverse
```

setgid and sticky bit

- setgid on file: set $EG_{roup}ID$ of process to GID of file
  on directory: sets group of files within

```
drwxrws---   4 root      adverse        4096 Feb 23  2018 adverse
```

- sticky bit

```
drwxrwxrwt 16 root      root            700 Feb  6 17:38 /tmp/
```

  - on: only file owner, directory owner, and root can rename or remove file in the directory
  - off: if user has write permission on directory, can rename or remove files, even if not owner

Resource isolation in UNIX

- **Pro:** Simple and flexible
- **Con:**
  - Coarse grained (not flexible enough)
  - Nearly all system operations require root access

From process isolation to sandboxing

- **Goal:** run untrusted code in a process
  - Process isolation is often not enough
  - Can attack the kernel, can access the filesystem, etc.
  - Need to restrict interface to/from untrusted code
- **Observation:** interface to outside world is the syscall interface
- **Idea:** monitor system calls and block unauthorized access

Seccomp BPF (SECure COMPuting with filters)

Introduction

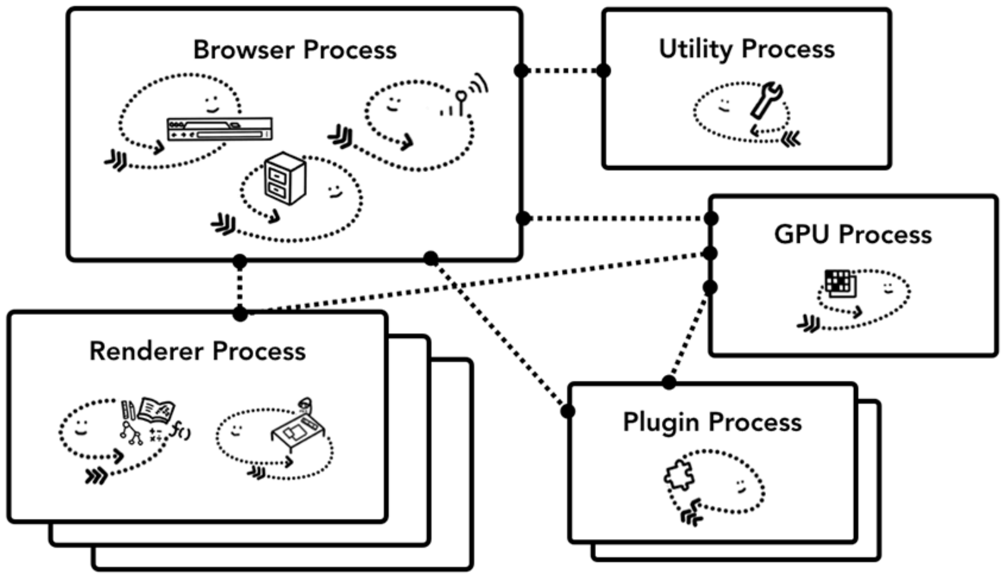
A large number of system calls are exposed to every userland process with many of them going unused for the entire lifetime of the process. As system calls change and mature, bugs are found and eradicated. A certain subset of userland applications benefit by having a reduced set of available system calls. The resulting set reduces the total kernel surface exposed to the application. System call filtering is meant for use with those applications.

Seccomp filtering provides a means for a process to specify a filter for incoming system calls. The filter is expressed as a Berkeley Packet Filter (BPF) program, as with socket filters, except that the data operated on is related to the system call being made: system call number and the system call arguments. This allows for expressive filtering of system calls using a filter program language with a long history of being exposed to userland and a straightforward data set.

Additionally, BPF makes it impossible for users of seccomp to fall prey to time-of-check-time-of-use (TOCTOU) attacks that are common in system call interposition frameworks. BPF programs may not dereference pointers which constrains all filters to solely evaluating the system call arguments directly.

Example: Modern browsers

- Browser process
  - ➤ Handles the privileged parts of browser (e.g., network requests, address bar, bookmarks, etc.)
- Sandboxed renderer process
  - ➤ Handles untrusted, attacker content: JS engine, DOM, etc.
  - ➤ Communication restricted to remote procedure calls (RPC)

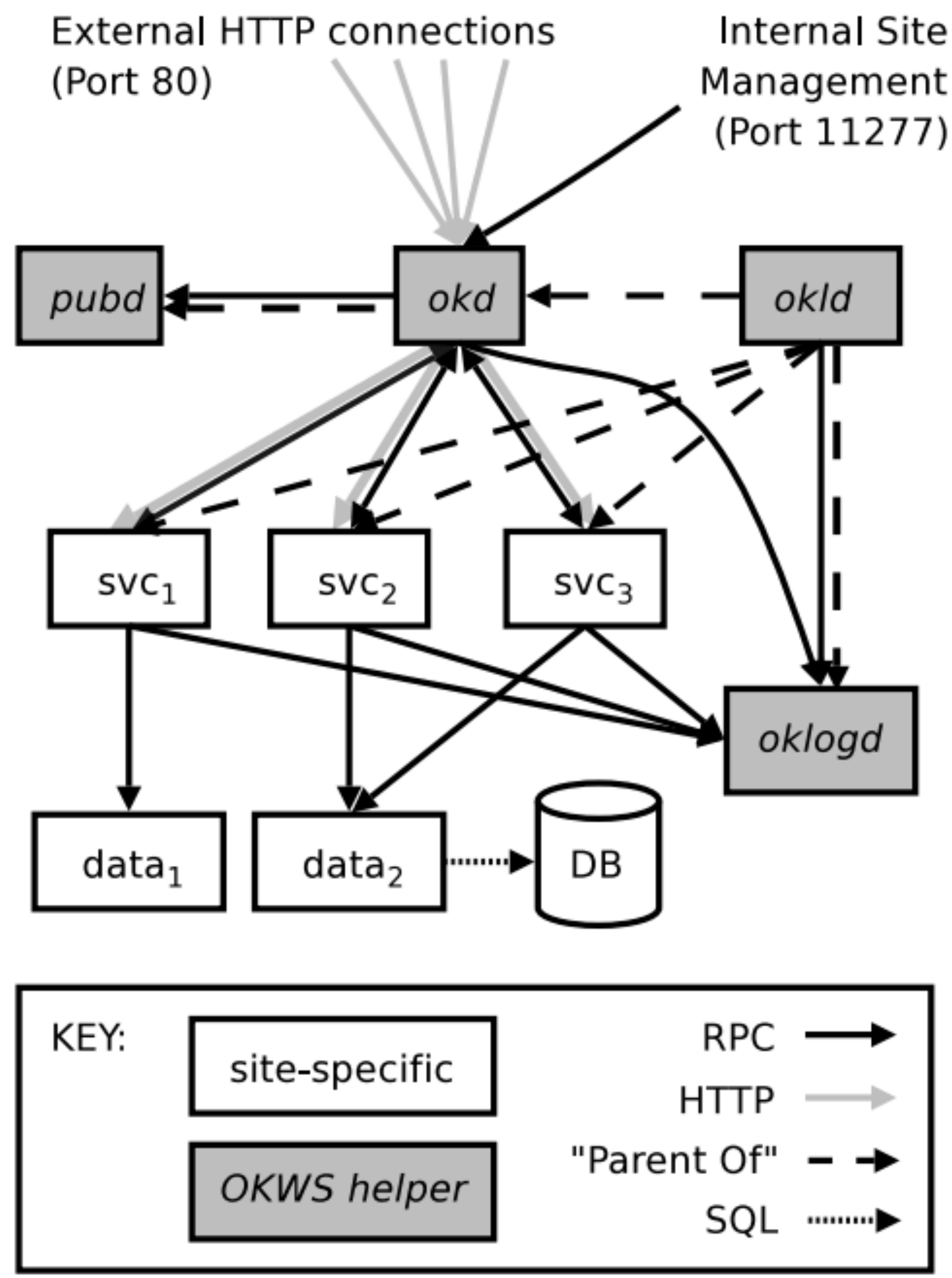


- Many other processes (GPU, plugin, etc)

https://developers.google.com/web/updates/2018/09/inside-browser-part1

Example: OK$_{\text{Cupid}}$W$_{\text{eb}}$S$_{\text{erver}}$

- Privilege separate services
  - ➤ Each service runs with unique UID
  - ➤ Memory + FS isolation
- Communication limited to structured RPC



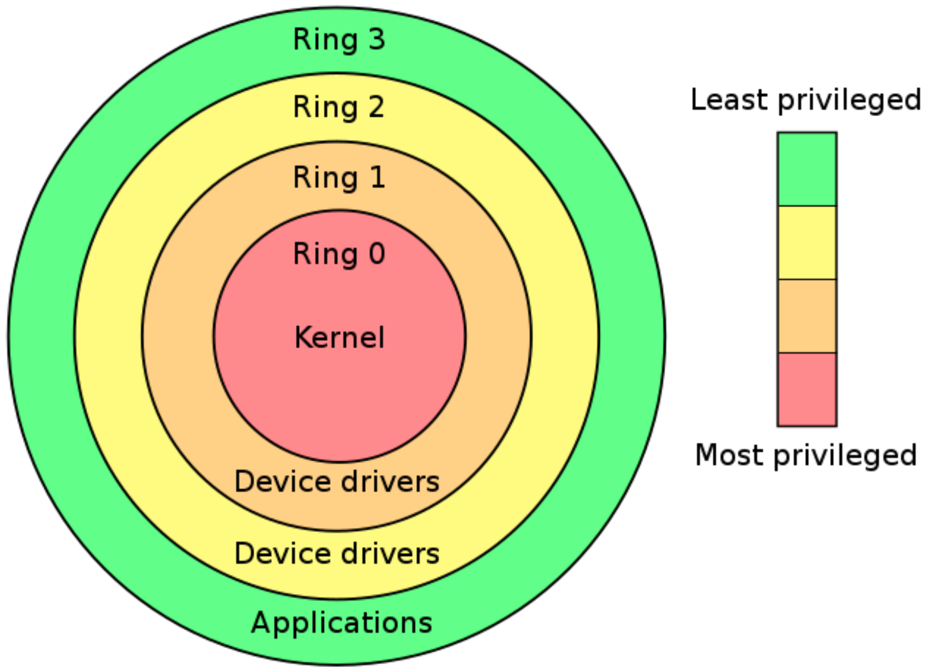
Example: Android

- Each app runs with own process UID
  - Memory + file system isolation
- Communication limited to using UNIX domain sockets + reference monitor checks permissions
  - User grants access at install time + runtime

Beyond process isolation...

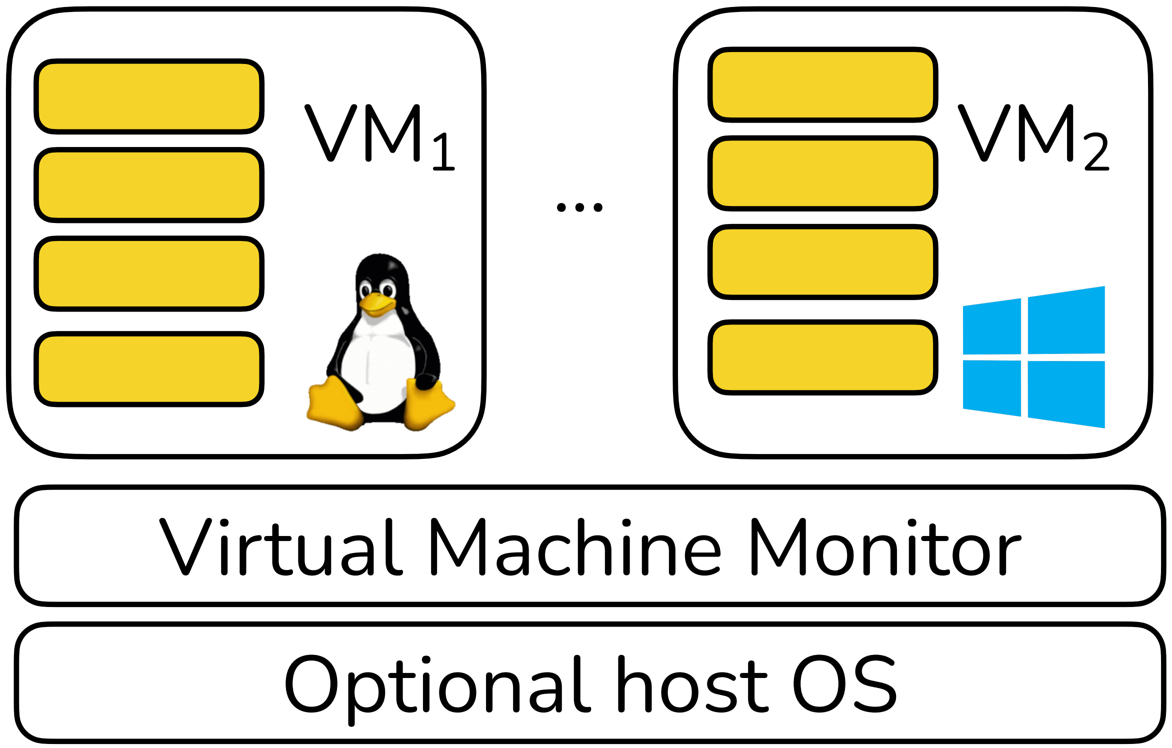
Example: Kernel isolation

- Kernel is isolated from user processes
  - Separate page tables
  - Processor privilege levels ensure userspace code cannot use privileged instructions
- Interface between userspace and kernel: syscalls



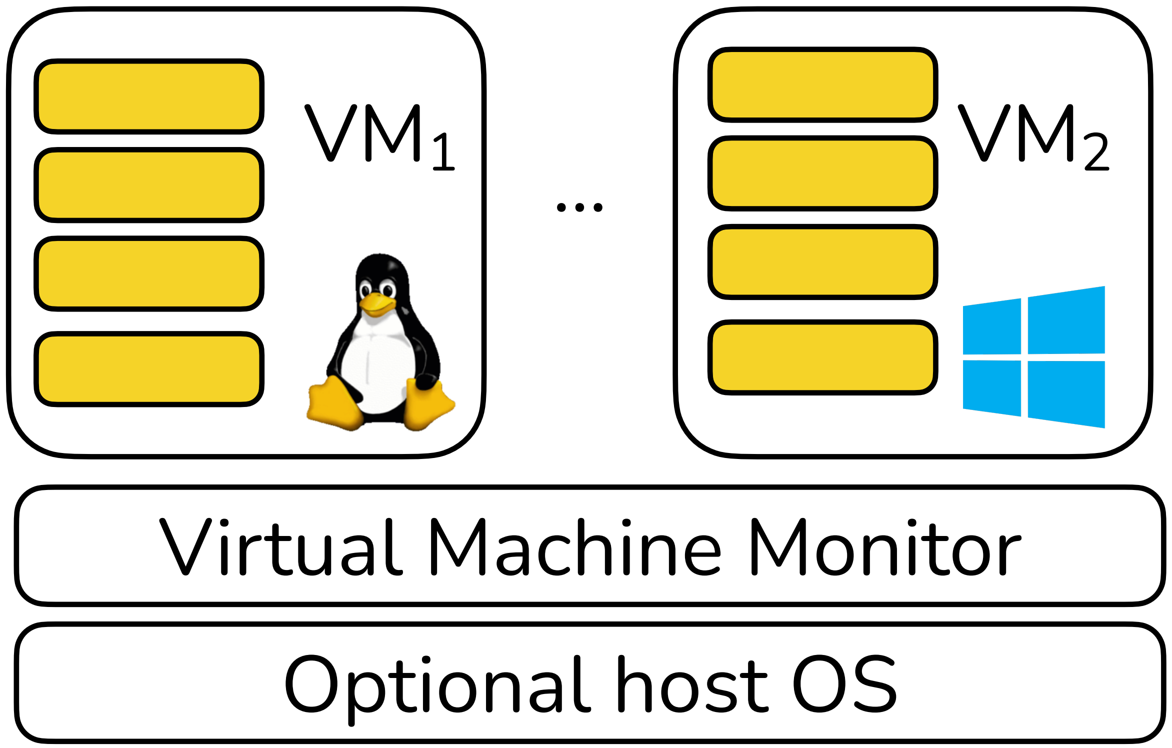
https://en.wikipedia.org/wiki/Protection_ring

Example: Virtual machines



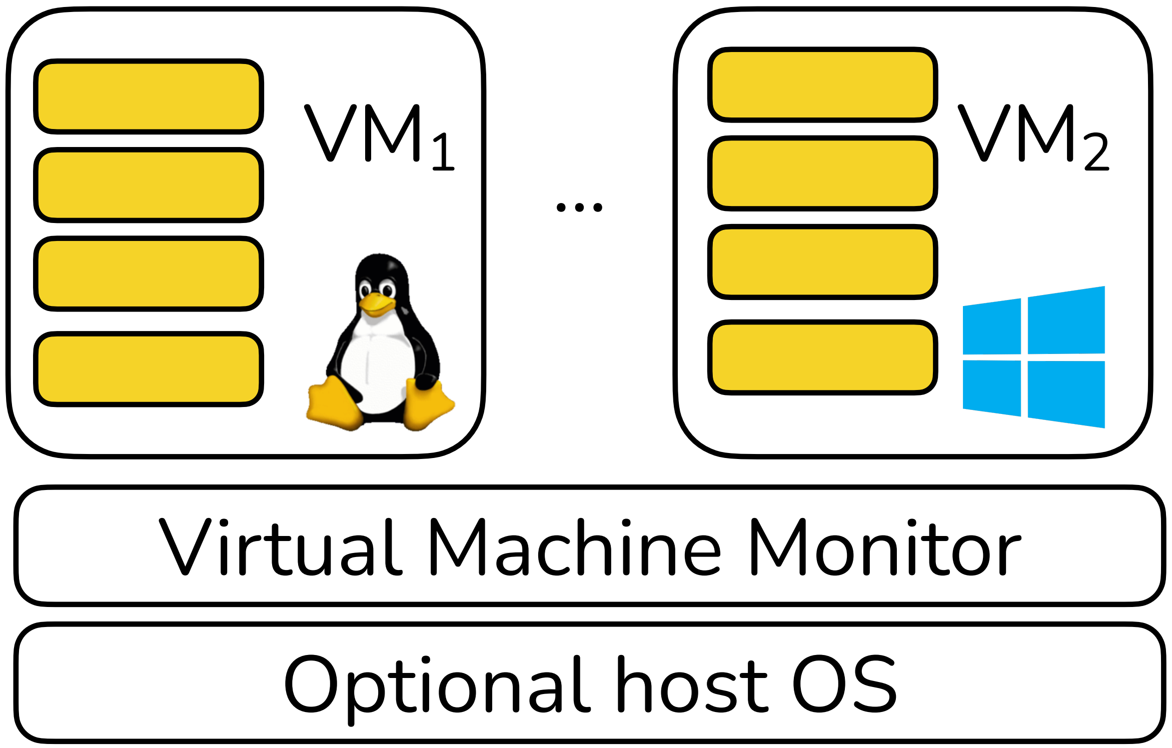
Example: Virtual machines

- Isolate VMs from each other
  - Nested page tables allows VM OS to map guest PA to machine PA



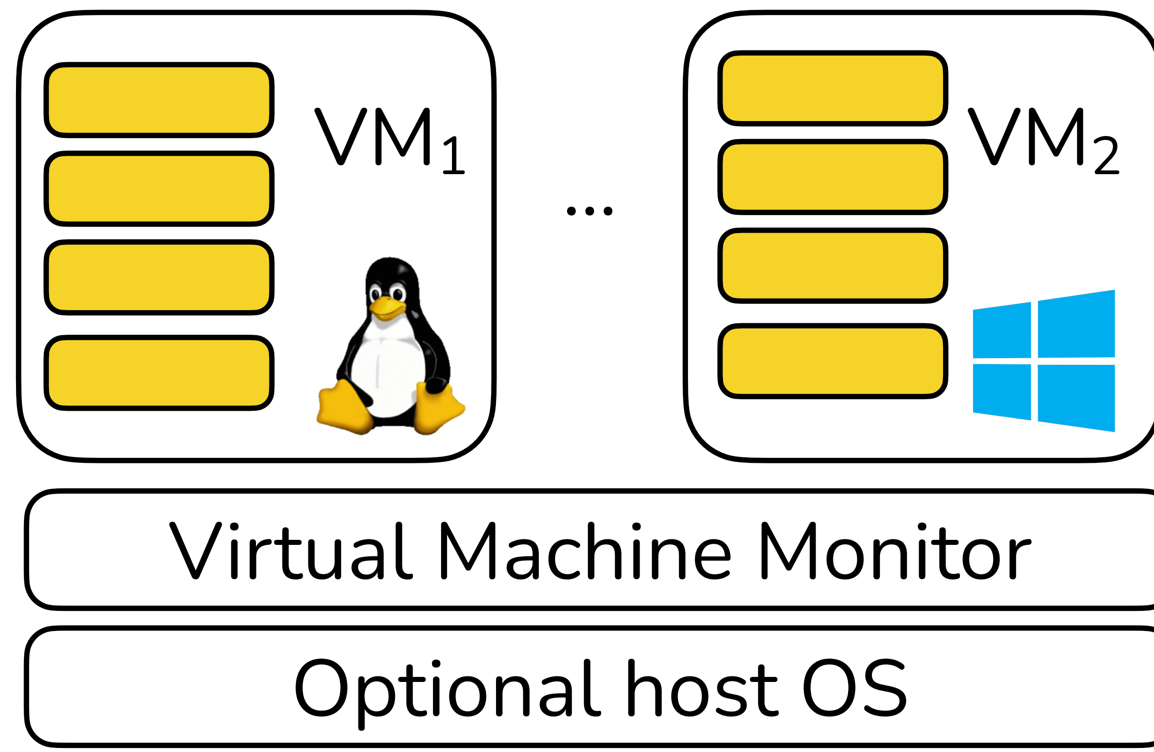
Example: Virtual machines

- Isolate VMs from each other
  - Nested page tables allows VM OS to map guest PA to machine PA
  - TLB entries are also tagged with VM ID (VPID)



Example: Virtual machines

- Isolate VMs from each other
  - ➤ Nested page tables allows VM OS to map guest PA to machine PA
  - ➤ TLB entries are also tagged with VM ID (VPID)
- Interface between VMs and VMM: hypercalls



Lots of isolation mechanisms

- Hardware-based isolation:
  - ➤ Physical machine, CPU modes (e.g., rings), virtual memory (MMU), memory protection unit (MPU), trusted execution environments, …
- Software-based isolation:
  - ➤ Language virtual machines (e.g., JavaScript), software-based fault isolation (e.g., WebAssembly), binary instrumentation, type systems, …

Software-based isolation

Why would we want to isolate things in software?

- Don’t have hardware-enforcement mechanism
- Process abstraction is too expensive
  - Startup latency, memory overhead, context switching

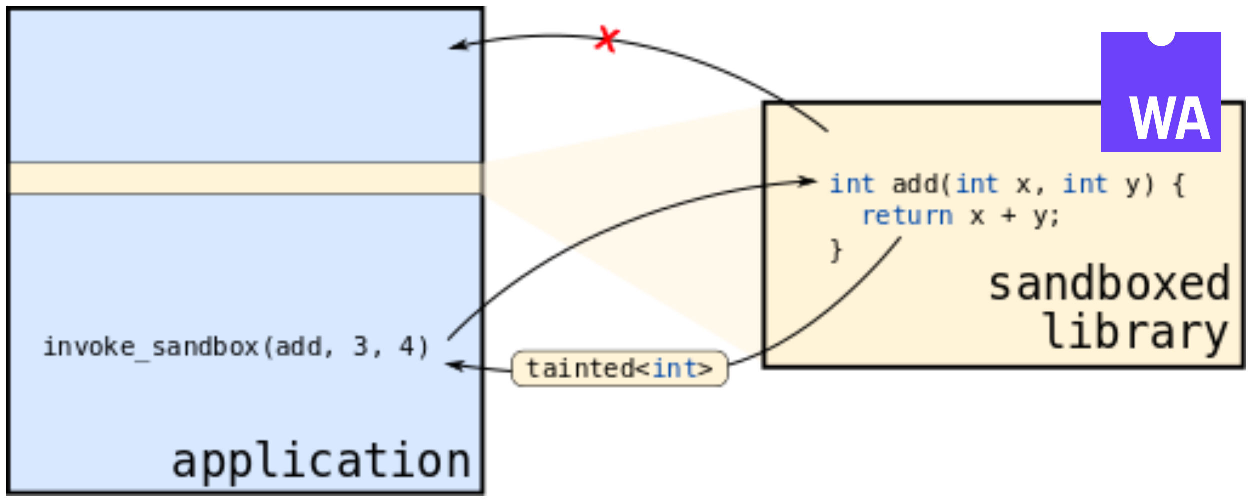
Software-based isolation

How can we isolate components in software?

- <u>Memory isolation:</u> instrument all loads and stores
- <u>Control flow integrity:</u> ensure all control flow is restricted to CFG that instruments loads/stores
- <u>Complete mediation:</u> disallow “privileged” instructions must crossing boundary to perform IO

Example: library sandboxing in Firefox

- Privilege separate renderer by isolating libraries
  - ➤ Why?
  - ➤ Isolation in software via WebAssembly
- Interface between libs and Firefox is typed



Example: library sandboxing in Firefox

- Privilege separate renderer by isolating libraries
  - Why?
  - Isolation in software via WebAssembly
- Interface between libs and Firefox is typed

Out of bounds memory write while processing Vorbis audio data

Announced March 16, 2018

Impact critical

Products Firefox, Firefox ESR

Fixed in Firefox 59.0.1
Firefox ESR 52.7.2

CVE-2018-5146: Out of bounds memory write in libvorbis

Reporter Richard Zhu via Trend Micro's Zero Day Initiative

Impact critical

Description

An out of bounds memory write while processing Vorbis audio data was reported through the Pwn2Own contest.

References

Bug 1446062

CVE-2018-5147: Out of bounds memory write in libtremor

Reporter Huzaifa Sidhpurwala

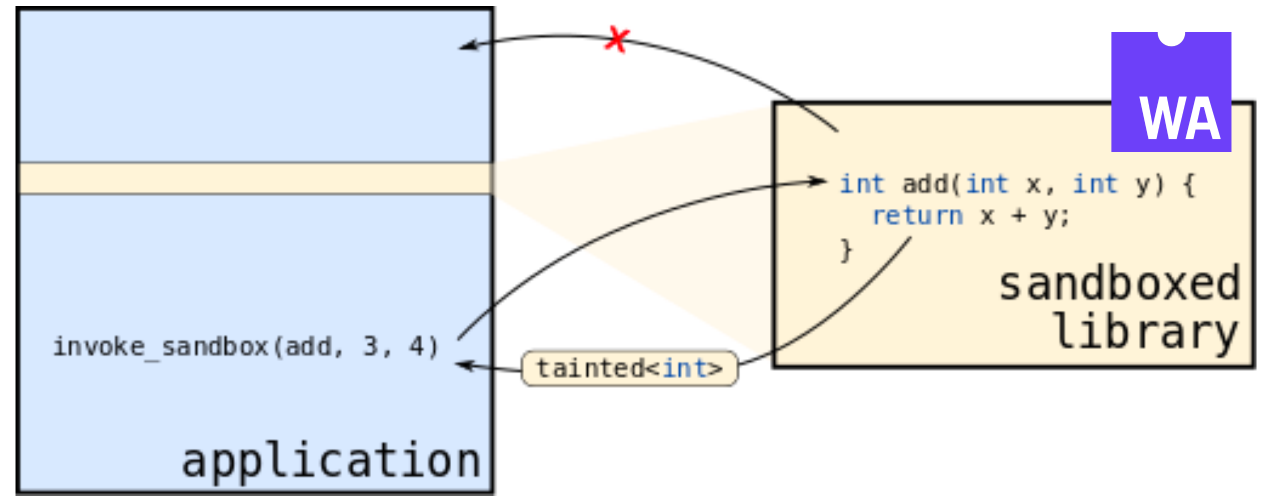
Impact critical

Description

The libtremor library has the same flaw as CVE-2018-5146. This library is used by Firefox in place of libvorbis on Android and ARM platforms.

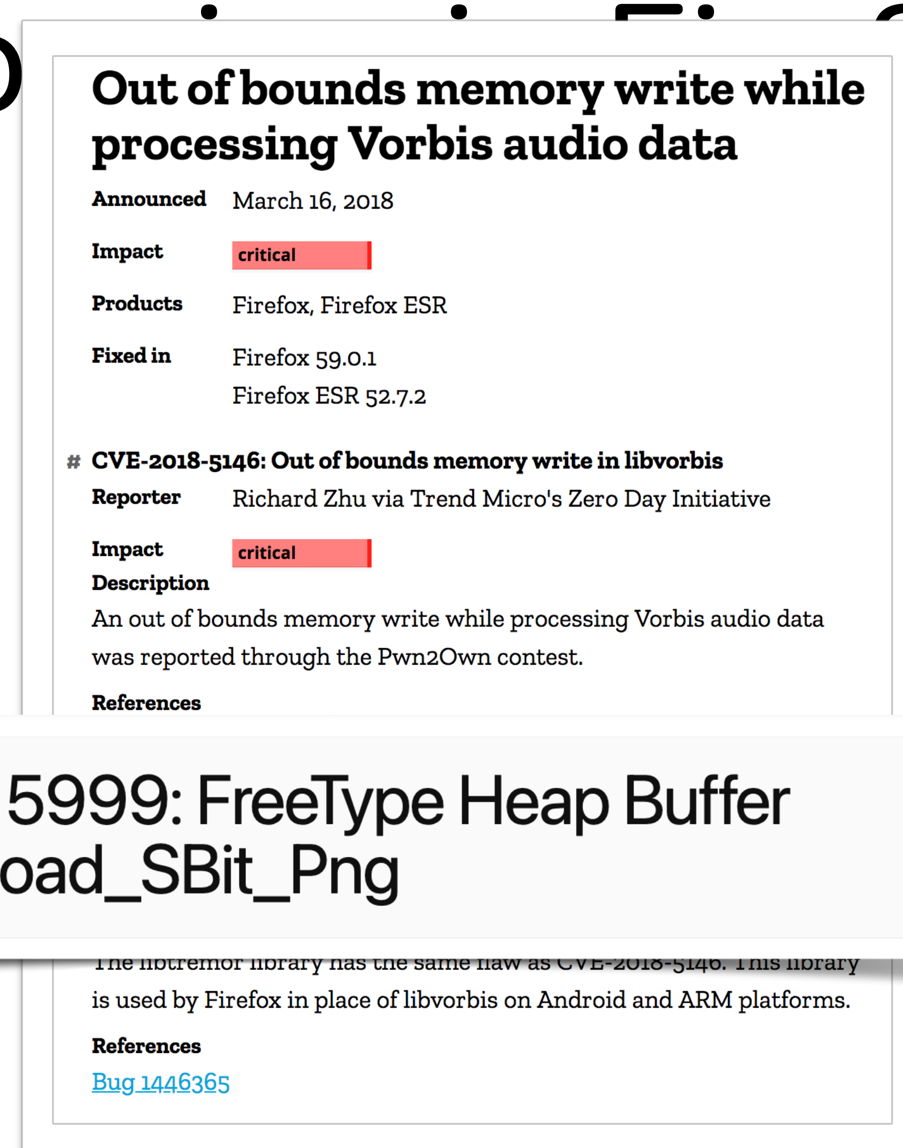
References

Bug 1446365



Example: library sandb ox

- Privilege separate renderer by isolating libraries
  - ➤ Why?
  - ➤ Isolation in software via WebAssembly
- Interface between libs and Firefox is typed



Out of bounds memory write while processing Vorbis audio data

Announced March 16, 2018

Impact critical

Products Firefox, Firefox ESR

Fixed in Firefox 59.0.1
Firefox ESR 52.7.2

CVE-2018-5146: Out of bounds memory write in libvorbis

Reporter Richard Zhu via Trend Micro's Zero Day Initiative

Impact critical

Description

An out of bounds memory write while processing Vorbis audio data was reported through the Pwn2Own contest.

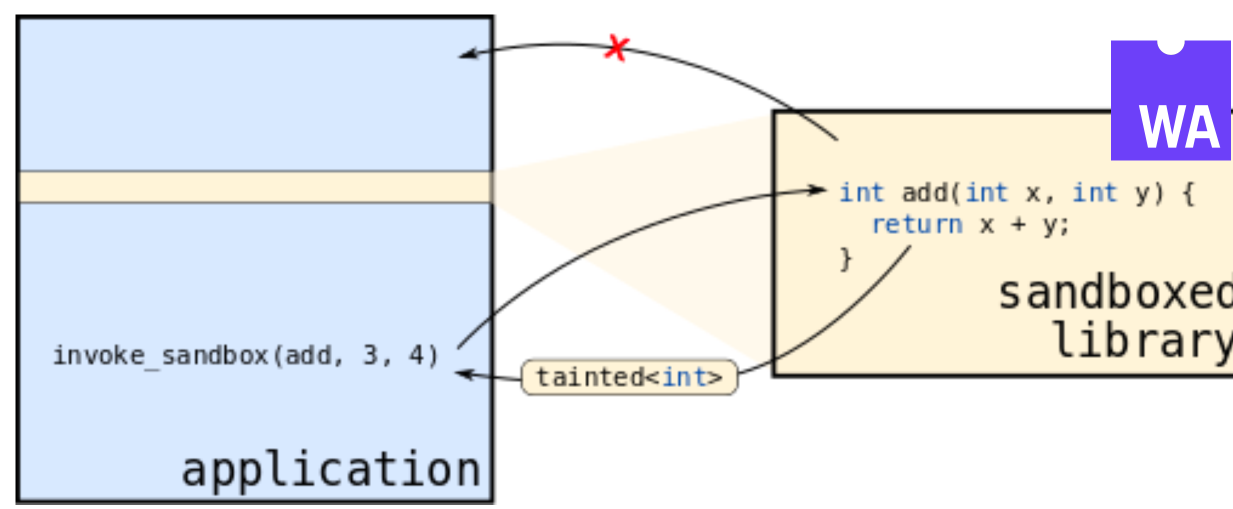
References

The libtremor library has the same flaw as CVE-2018-5146. This library is used by Firefox in place of libvorbis on Android and ARM platforms.

References

Bug 1446365

CVE-2020-15999: FreeType Heap Buffer Overflow in Load_SBit_Png



Isolation is not enough!

- Why not?
  - To do anything useful we typically need to cross trust boundaryIsolation is not enough
  - E.g., syscalls, hypercalls, runtime calls
- Need to ensure that the *calls are correct
  - Must keep track of whether you're operating on untrusted data or not
  - Incorrect implementations -> confused deputy attacks

Example: library isolation in Firefox

```
void create_jpeg_parser() {

 jpeg_decompress_struct jpeg_img;
 jpeg_source_mgr        jpeg_input_source_mgr;


 jpeg_create_decompress(&jpeg_img);
 jpeg_img.src = &jpeg_input_source_mgr;
 jpeg_img.src->fill_input_buffer = /* Set input bytes source */;


 jpeg_read_header(&jpeg_img /* ... */);
 uint32_t* outputBuffer =  /* ... */;

 while (/* check for output lines */) {
    uint32_t size = jpeg_img.output_width * jpeg_img.output_components;

    memcpy(outputBuffer, /* ... */, size);
 }
}
```

Can we make this easier? (kernel)

- Explicit functions for copying data:
  - `copy_to_user()` and `copy_from_user()`
- HW to prevent kernel from accessing user data
  - ARM Privilege Access Never/Privileged eXecute Never
- Support for limiting and filtering system calls
  - E.g., browsers use seccomp-bpf to restrict the syscall interface of untrusted processes (and thus pwnage via kernel exploitation)

Can we make this easier? (browser)

- Restrict interface to RPC
  - ➤ Generate RPC interface from interface description languages
  - ➤ RPC ensure type and memory safety
- Tainted types (RLBox)
  - ➤ Eliminate confused deputy attacks by forcing trusted code to validate all untrusted data before using it

Example: library isolation in Firefox

```
void create_jpeg_parser() {

 auto sandbox = rlbox::create_sandbox<wasm>();
 tainted<jpeg_decompress_struct*> p_jpeg_img = sandbox.malloc_in_sandbox<jpeg_decompress_struct>();
 tainted<jpeg_source_mgr*>        p_jpeg_input_source_mgr = sandbox.malloc_in_sandbox<jpeg_source_mgr>();

 sandbox.invoke(jpeg_create_decompress, p_jpeg_img);
 p_jpeg_img->src = p_jpeg_input_source_mgr;

 p_jpeg_img->src->fill_input_buffer = /* Set input bytes source */;

 sandbox.invoke(jpeg_read_header, p_jpeg_img /* ... */);
 uint32_t* outputBuffer =  /* ... */;

 while (/* check for output lines */) {
    uint32_t size = (p_jpeg_img->output_width *  p_jpeg_img->output_components).copy_and_verify(
        [](uint32_t val) -> uint32_t {
           assert(val <= outputBufferSize);
           return val;
        });
    memcpy(outputBuffer, /* ... */, size);
 }
}
```

Principles of secure design

- Least privilege
- Privilege separation
- Complete mediation
- Fail safe/closed
- Keep it simple
- Defense in depth

Principles of secure design

- Fail safe/closed
  - ➤ Why might system designers choose to fail open?
  - ➤ What's the problem with failing open?
- Keep it simple
  - ➤ Complexity is the enemy of security!
- Defense in depth
  - ➤ Don't expect defenses to be perfect

Today

Lecture objectives:

- Understand basic principles for building secure systems
- Understand mechanisms used to build secure systems