KVM FORUM

Securing the Hypervisor with Control-Flow Integrity

Daniele Buono
IBM Research

THE LINUX FOUNDATION

Special Thanks

- Team @ IBM Research:

- Hubertus Franke
- James Bottomley
- Tobin Feldman-Fitzthum
- Dov Murik
- James Cadden
- Marcio A Silva
- Carlo Bertolli

- QEMU Community

- Paolo Bonzini
- Philippe Mathieu-Daudé
- Daniel P. Berrangé
- Stefan Hajnoczi
- Wainer dos Santos Moschetta

Outline

- Why Control-Flow Integrity
- Implementing Control-Flow Integrity
  - Stack Protection
  - Function Pointers Protection
- Status of the patches
- What did we accomplish?
- Future Work

Why Control-Flow Integrity

QEMU, as every other computer program, is subject to bugs

Vulnerability Trends Over Time[1]

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Directory Traversal	Bypass something	Gain Information	Gain Privileges
2007	4		1	2					
2008	5	1						1	
2009	1		1						
2010	1	1	1	1					
2012	7	3	1	2			1		3
2013	4	2	1	2					1
2014	36	17	27	24	4			2	2
2015	11	8	6	7	1			1	1
2016	91	65	10	17	2	1		8	
2017	65	55	6	12	1			1	2
2018	38	12	6	10	1				
2019	17	3	4	3			1		
2020	46	16	6	12		1	1		
2021	25	13	6	6			1	2	
Total	351	196	76	98	9	2	4	15	9
% Of All		55.8	21.7	27.9	2.6	0.6	1.1	4.3	2.6

Integrity Attacks:
Get access to:

- Provider infrastructure
- Other Clients
  - Data
  - VM

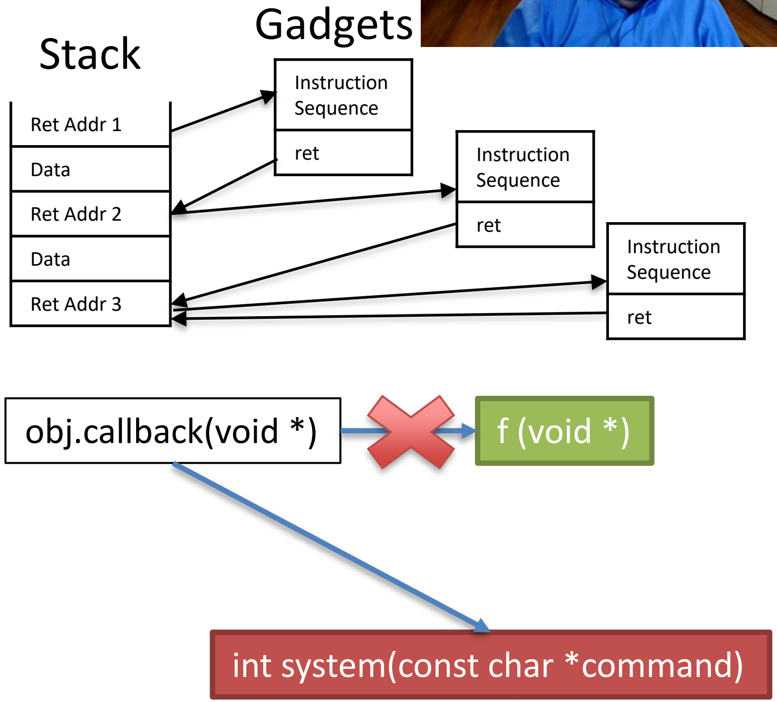
[1] data gathered from www.cvedetails.com

`int system(const char *command);`

Why Control-Flow Integrity

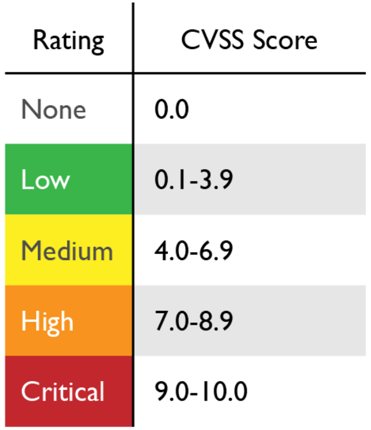
Typical Techniques for Integrity Attacks:

- ROP Gadgets
  - With a buffer overflow, an attacker can overwrite the stack to call a sequence of ROP Gadgets (ROP Chain)
- Indirect Function Call Hijacking:
  - Modify function pointers in data structures, to change the behavior of the program
  - Very effective in C++ code (Virtual Functions) or C code with callbacks
  - Usually, insert calls to libc that translate into syscalls (mmap, mprotect, system, …)



Why Control-Flow Integrity

How effective are vulnerabilities in QEMU, anyway?



Rating	CVSS Score
None	0.0
Low	0.1-3.9
Medium	4.0-6.9
High	7.0-8.9
Critical	9.0-10.0

Data from 2017:

Vulnerabilities with a CVSS score

- >= 4: 79 (41%)
- >= 7: 14 (4%)

But surely, an attack with code execution scores at least "high", right?

[1] data gathered from www.cvedetails.com

Why Control-Flow Integrity

Surely, an attack with code execution scores at least "high"

- Turns out, understanding the impact of a vulnerability is not that easy
  - CVE-2019-14378:
    ip_reass in ip_input.c in libslirp 4.0.0 has a heap-based buffer overflow via a large packet because it mishandles a case involving the first fragment.

CVSS Score	**6.5**
Confidentiality Impact	Partial (There is considerable informational disclosure.)
Integrity Impact	Partial (Modification of some system files or information is possible, but the attacker does not have control over what can be modified, or the scope of what the attacker can affect is limited.)
Availability Impact	Partial (There is reduced performance or interruptions in resource availability.)
Access Complexity	Low (Specialized access conditions or extenuating circumstances do not exist. Very little knowledge or skill is required to exploit.)
Gained Access	None

THE LINUX FOUNDATION

Why Control-Flow Integrity

Surely, an attack with code execution scores at least “high”

- Turns out, understanding the impact of a vulnerability is not that easy

  – CVE-2019-14378: https://vishnudevtj.github.io/notes/qemu-vm-escape-cve-2019-14378
    - Bug in packet fragment reassembling. Can write to an offset from an allocated packet.
    - Spray Heap by sending a fragmented packet with no end fragment, gain control of the heap
    - Sending specific sequence of fragmented packets, get arbitrary write
    - Use ICMP echo request to read memory, bypass ASLR and PIE. Find beginning of code section and data.
    - Overwrite QEMUTimerList pointer to custom-made callback list
    - Call System()

    - Similar attack with CVE-2019-6778 (CVSS 4.6): https://github.com/0xKira/qemu-vm-escape

Why Control-Flow Integrity

Surely, an attack with code execution scores at least "high"

- Turns out, understanding the impact of a vulnerability is not that easy
  - Attacks can use multiple vulnerabilities to obtain a larger effect.
  - CVE-2015-5165 and CVE-2015-7504

CVE-2015-5165		CVE-2015-7504	
CVSS Score	**5.0**	CVSS Score	**4.6**
Confidentiality Impact	Partial (There is considerable informational disclosure.)	Confidentiality Impact	Partial (There is considerable informational disclosure.)
Integrity Impact	None (There is no impact to the integrity of the system)	Integrity Impact	Partial (Modification of some system files or information is possible, but the attacker does not have control over what can be modified, or the scope of what the attacker can affect is limited.)
Availability Impact	None (There is no impact to the availability of the system.)	Availability Impact	Partial (There is reduced performance or interruptions in resource availability.)
Gained Access	None	Gained Access	None

THE LINUX FOUNDATION

Why Control-Flow Integrity

Surely, an attack with code execution scores at least “high”

- Turns out, understanding the impact of a vulnerability is not that easy
  - Attacks can use multiple vulnerabilities to obtain a larger effect.
  - CVE-2015-5165 and CVE-2015-7504: http://www.phrack.org/papers/vm-escape-qemu-case-study.html
    - Use CVE-2015-5165 to read memory that was previously used for setup objects in QEMU – memory freed but not cleared.
    - Find a specific object with a function pointer
    - Use such pointer to compute known function and data addresses
    - Use CVE-2015-7504 to change virtual irq function pointer to mprotect: make guest mem executable on host
    - Use CVE-2015-7504 again to change virtual irq function pointer to the executable guest memory
    - Run arbitrary code on the host. Example: a shell with stdin and stdout inside the VM

Why Control-Flow Integrity

~~Surely, an attack with code execution scores at least "high"~~

- Turns out, understanding the impact of a vulnerability is not that easy
- It should be assumed that any buffer overflow (no matter the CVSS score) can result in an arbitrary memory write ***and*** compromise the host.
- Buffer overflow on the Stack can also be used for ROP attacks – however QEMU allocates most structures on heap so Stack Overflows are less likely.
- CVEs since 2015: 5% stack-based buffer overflow[1], 31% buffer overflow[2]

[1] CVEs with "stack" in the description. [2]CVEs with "buffer overflow" and "out-of-bounds" in the description.

Why Control-Flow Integrity

QEMU, as every other computer program, is subject to bugs

- Solutions?
  - Remove bugs!
    - Updating QEMU means stopping the VM – Hurting Availability
    - Only affect known bugs.
  - Avoid bugs by using “safe” languages?
    - Safe languages only help avoiding some types
    - Impractical on 2.5+ Millions of lines of code
    - Needs a total rewrite – other projects are pursuing that: Firecracker, Cloud Hypervisor…
  - Reduce effectiveness of bugs
    - Seccomp, SELinux, AppArmor, cgroups, namespaces, etc.
      - Usually end up with a “loose protection”, in order to cover all the types of behavior QEMU have.
    - Control Flow Integrity
      - Inside QEMU’s binary, have better chances to stop unwanted behaviors

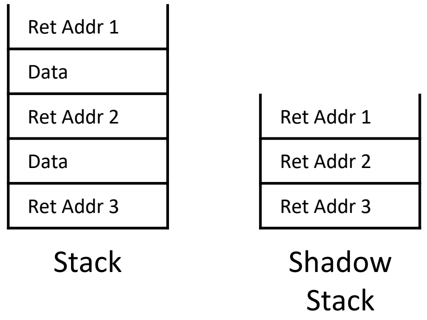
Implementing Control-Flow Integrity

- Google's Chromium and Android are the gold standard for advanced security features in C/C++ code
  - Approach: add advanced, research-based features in the compiler
    - Generic: can be used with *"little"* effort on multiple projects
  - Compiler of choice: Clang/LLVM
    - Backward-Edge Control Flow Integrity (ShadowCallStack, SafeStack)
    - Forward-Edge Control-Flow Integrity (icall CFI)
    - Hardened Memory Allocator (SCUDO) – Statistically Identify buffer overflows
    - Undefined Behavior checks – Integer overflow, Array out of Boundary, etc.
- Compiler Based? We can use the same features in QEMU!
  - With *"small"* changes, and lots of testing

Protecting the Stack – Shadow Stack

The standard protection against Stack Smashing

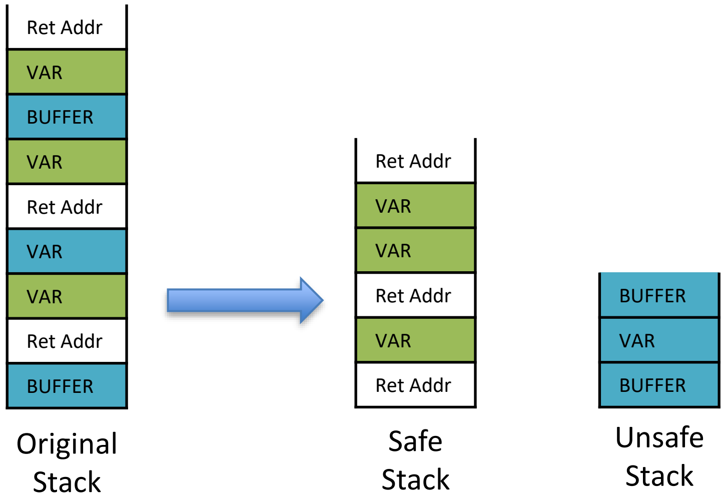
- Idea: Keep a copy of all the return pointers somewhere else
  - At every call, push return address to Shadow Stack
  - At every return, pop address from Shadow Stack and check
  - Can be implemented at Software or in Hardware
- Caveats:
  - Software version insecure in some architectures
    - Race conditions, a thread can change return address before push/after check
    - Additional instructions on call/ret reduce performance
  - Hardware versions such as CET fix most issues
    - Avoid Race condition, value is saved while executing call, and checked while executing ret
    - Can protect memory effectively, since shadow stack is only accessed by call/ret instructions



Protecting the Stack – SafeStack

SafeStack is an alternative protection to stack smashing

- Idea: Instead of copying the return pointer into a shadow position,
  - Let's make sure that a buffer overflow cannot overwrite a return pointer!
- How:
  - The stack is split in "Safe" and "Unsafe" stack.
  - "Safe" stack only contains data accessed with fixed-length operations
  - Everything that may cause a buffer overflow, is stored in the "Unsafe" Stack



Shadow Stack vs SafeStack

- SafeStack is safer and faster than software-based Shadow Stacks
  - Safety:
    - No race conditions that can void protection
  - Performance:
    - No need to add instructions around call/ret
    - Only need an additional register to store the unsafe stack pointer (Shadow Stack also need a register for second stack pointer)
    - There will be caching differences (data that was contiguous in a normal stack, may not be with SafeStack), which may hurt performance in highly optimized stacks.
- SafeStack should be comparable to hardware-based Shadow Stacks
  - Safety:
    - No race conditions that can void protection in both cases
    - However, some hardware-based shadow stacks (CET) can protect the shadow stack memory
  - Performance:
    - Since SafeStack has almost no overhead, performance should be comparable (if not better) to CET, which requires the execution of more complex instructions in place of call/ret

THE LINUX FOUNDATION

Implementing SafeStack in QEMU

- Add a flag, and you're good to go…in most cases
  - QEMU makes heavy use of coroutine[1] (mostly block layer)
    - Coroutines in QEMU are implemented using ucontext or sigaltstack, which are not supported by SafeStack
    - Each coroutine carries its own state – including stack
      - With SafeStack, we should:
        1. Create both safe and unsafe stack
        2. On a coroutine switch, both safe and unsafe stack need to be updated.
    - Updated ucontext-based implementation to support SafeStack
      - ucontext library is used only to "set-up" a co-routine
        - » Create new context <- Allocate unsafe stack too
        - » Swap to context first time <- Update unsafe stack pointer before context swap
      - Then, sigsetjmp/siglongjmp are used to switch between coroutines
        - » These are automatically supported by SafeStack and did not need additional changes

[1]Coroutines can be seen as lightweight, cooperative threads

Protecting Function Pointers

Forward-edge Control-Flow Integrity

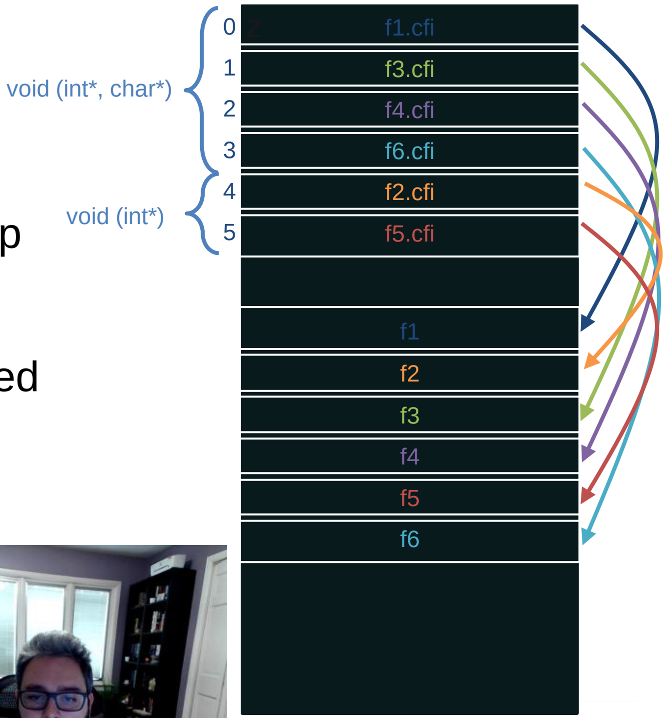
- Idea: Check that the function pointed is “allowed”
  - Theory: Call Graph can be used for exact matching
  - Practice: Call Graph is impracticable; use approximations
    - Intel CET IBT/Microsoft CFG: every function in the binary *that could be used as an indirect call* is allowed
    - Clang icall/Microsoft XFG: every function in the binary *that has the correct signature* is allowed

- How:
  - Ordered Jump Tables (Clang icall)
    - Pointers allowed only to jump tables
    - If pointer is within the correct interval, pass check
  - Signature Hashes (Clang icall/Microsoft XFG)
    - Compute hash of signature, store somewhere safe (.text)
    - If hash of function is correct, pass check
  - Add special instruction at beginning of function (Intel CET IBT)
    - Hardware will make sure that such instruction is encountered after executing *call*
- Signature checking clearly works at a finer grain

icall CFI implemented by Clang/LLVM

Ordered Jump Tables: checking signatures in constant time

- For every function, create a “shadow function”
  - The shadow function only contains a hardcoded jump to the real function
  - Shadow functions are ordered by signature type
  - Compiler makes sure that the shadow function is used when assigning function pointers
- Now, at call time, signature can be checked by making sure the pointer is in the memory area containing the right shadow functions
  - 0-3: void (int*, char*)
  - 4-5: void (int*)



Implementing icall CFI in QEMU

Is QEMU respecting function signatures,
i.e. no weird type casting happening under the hood?

- Yes… But
  - Several cases where registered callbacks have different pointer types:
    - Callback pointer is a void (int*), callback type is void (char*)

- Clang offers a useful option to consider all pointers equal (pointer generalization)
  - void (int*) == void (char*))

THE LINUX FOUNDATION

Implementing icall CFI in QEMU

- Again, add a flag, and you're good to go…in most cases
  - QEMU has several "sensitive points", where icall CFI has issues, because the function called does not have a shadow function:
    - Anything that uses JIT compilation, where the compiled binary is allocated in memory at run-time
    - Anything that uses a callback that is coming from outside of the binary (i.e. shared libraries)
  - In particular, we have issues with:
    - TCG: Disable CFI when calling Translation Blocks
    - TCI: Disable CFI when interpreting instructions
    - Plugins: Disable CFI for plugin callbacks
    - Modules: Block CFI at compile time if modules are enabled
    - QEMU Signal Handler[1]:Disable CFI for the signal handler call

[1]QEMU can call the installed signal handler directly in some cases

Implementing icall CFI in QEMU

Bonus side effect:

- icall CFI requires the use of Link-Time Optimization (LTO), to create and reorder shadow functions for the entire binary at once (not per-object file)
  - Tested and added support for LTO in QEMU, using Meson

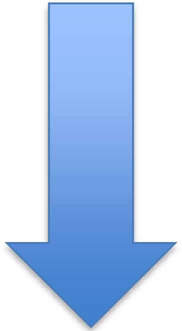
Status of patches

- Both SafeStack and CFI icall patches are upstream
- Reduced CI case running in gitlab to check against bit rot
  - LTO significantly increase compile time and memory requirements for compilation.
    - Shared gitlab runners only allow limited testing
- Features can be used today, with some caveats
  - Requires Clang.
    - Most distributions use GCC, have to compile your own QEMU
  - Icall CFI does not work with modules
    - Some distribution (i.e. RHEL) are moving towards a modular build, which will limit its adoption in current form

THE LINUX FOUNDATION

Did we accomplish something?

CVEs since 2015: 5% stack-based buffer overflow[1], 31% buffer overflow[2]



Mitigated Approximately 35% of known CVEs[3], + Zero-Day Vulnerabilities

[1]CVEs with "stack" in the description. [2]CVEs with "buffer overflow" and "out-of-bounds" in the description.
[3]CVEs with “stack”, "buffer overflow" and "out-of-bounds" in the description.

What’s next?

- Support for CFI in shared libraries
  - Necessary for:
    - Enabling CFI with modules
    - Enabling CFI with shared library version of slirp
- Problem:
  - Jump Tables, and address windows, are computed on the binary
  - A shared library, even if instrumented with CFI, will have its own jump table and the address window for the same signature will be disjoint from the main binary.
- Solution:
  - Clang implements Cross-DSO CFI
    - If “local” CFI fails, take slower path to check CFI in external DSO
      - If external library was instrumented, call is protected
      - If external library was not instrumented, call is not protected <- Changes in LLVM
    - Additional performance penalty if “local” CFI fails <- Shouldn’t happen often
    - Not compatible with pointer generalization <- Changes in QEMU
    - Experimental implementation with performance and correctness issues in dlopen
      - Would need implementation in libc or libdl for production <- Changes in libc

KVM FORUM