ENABLE INTEL CET IN LINUX OS

H.J. Lu

Intel

August 2020

Introduction

Control-flow Enforcement Technology (CET)

- An upcoming Intel® processor feature that blocks return/jump-oriented programming (ROP) attacks
- Two components:
  - Shadow Stack (SHSTK)
  - Indirect Branch Tracking (IBT)

Control-flow Definition

The code execution path, branched by RET, JMP, or CALL.

Op Code	Operand
RET	On program stack
JMP *%rax	In memory (%rax as a pointer)
CALL *%rax	In memory (%rax as a pointer)

Shadow Stack Management

- Most of programs are compatible with SHSTK. No special treatment is needed.
- setjmp and longjmp
  - Save shadow stack pointer. Increment shadow stack pointer until the old value is restored.
- ucontext
  - Allocate a new shadow stack for each user provided stack with a restore token.
  - Use restore token to switch shadow stack.

Adjust Shadow Stack

Adjust shadow stack if return address on normal stack isn't the same as the one on shadow stack.

- If the number of stake frames skipped is known, increment shadow stack pointer by the number of stack frames skipped.
- Otherwise, pop shadow stack one frame at a time until return address on normal stack matches the one on shadow stack.

Indirect Branch Tracking

- All indirect branch targets must start with ENDBR64/ENDBR32.
  - ENDBR64/ENDBR32 is NOP on non-CET processors.
- The “notrack” prefix before indirect branches disables IBT.

Legacy Bitmap

CET processor supports the optional legacy bitmap. IBT is disabled on pages in the legacy bitmap. However, the legacy bitmap doesn't cover the legacy JIT engines:

- When loading a legacy shared object which contains a legacy JIT engine, loader puts the shared object into the legacy bitmap.
- The legacy JIT engine allocates a page to place the legacy jitted codes, which isn't covered by the legacy bitmap.
- The application crashes when the legacy jitted codes are executed.

Solution: Don't use legacy bitmap and treat IBT like SHSTK when loading a legacy shared object.

Enable CET on Linux

- Enable CET on Linux is equivalent to porting Linux to a new architecture.
  - Only CET enabled Linux on CET processors can provide CET security.
  - Every piece of OS must be CET enabled, starting from kernel, toolchain, libraries, …
    - A binary is CET enabled only if all its components are CET enabled.
- CET enabled OS is backward compatible.
  - The same CET-enabled OS binary can run on CET and legacy processors.
    - Provide CET security only on CET processors.
    - No performance loss on legacy processors.

Linux CET Run-time

At run-time, kernel starts loader of a dynamic application with CET is enabled. Loader disables CET if any loaded shared objects aren't CET enabled. Loader issues an error when dlopening a legacy shared object from a CET-enabled process.

- Configure time option, --enable-cet=permissive:
  - Disable CET when dlopening a legacy shared object from a CET-enabled process.
- Run-time control (Not applicable on SUID binaries):
  - GLIBC_TUNABLES=glibc.cpu.x86_ibt=permissive
  - GLIBC_TUNABLES=glibc.cpu.x86_shstk=permissive
    - Disable CET when dlopening a legacy shared object.
  - GLIBC_TUNABLES=glibc.cpu.x86_ibt=off
  - GLIBC_TUNABLES=glibc.cpu.x86_shstk=off
    - Always disable CET.
  - GLIBC_TUNABLES=glibc.cpu.x86_ibt=on
  - GLIBC_TUNABLES=glibc.cpu.x86_shstk=on
    - Never disable CET.

Enable CET in GCC Toolchain

- -fcf-protection with GCC:
  - Place ENDBR at all potential indirect branch targets.
  - Unwind shadow stack for stack unwind intrinsics.
  - Generate CET marker when CET is enabled.
  - Provide a header file, <cet.h>, to generate CET marker in assembly codes.
  - C, C++ and Fortran only.
- Linker:
  - Properly mark programs as CET enabled when all its components are marked as CET enabled.
    - Place ENDBR at all linker generated indirect branch targets

Enforce CET Marker On An Application

- A program/library is CET enabled only if all its components are CET enabled.
- Build program/library with the linker switch, -z cet-report=error, identifies input objects with missing CET marker:

```
$ gcc -Wl,-z,cet-report=error x.o
/usr/bin/ld: x.o: error: missing IBT and SHSTK properties
collect2: error: ld returned 1 exit status
```

Add CET Marker On An Object File

- If for some reason, there is no CET marker on the object file, but otherwise the object file is CET compatible, linker can add the CET marker on such file:
  - Source codes aren't available.
  - Tools used to generate object file can't generate the proper CET marker.
    - nasm
    - yasm
- "ld -r -z ibt –z shstk –o output-object input-object"
  - Linker will add the CET marker to "output-object".

LLVM

- CET is functional in LLVM 11.
  - CET fixes have been backported to LLVM 10.x release:
    - Enable CET in C++ exception handling.
    - Enable CET in JIT.
    - <cet.h> is only available in LLVM 11.
- Libunwind isn't CET enabled:
  - https://bugs.llvm.org/show_bug.cgi?id=45946

Enable CET in Applications

- For C/C++/FORTRAN sources, just compile with -fcf-protection.
  - For JIT:
    - Add ENDBR at indirect branch targets.
    - Adjust shadow stack when stack frames are skipped.
- For assembly sources:
  - Place ENDBR at all potential indirect branch targets in assembly codes.
  - Mark all assembly codes as CET enabled.
    - Include <cet.h> if possible.
- Other high level languages aren't CET yet.

Software Status

- CET has been enabled in Linux toolchain.
  - GCC 8 or above.
  - Binutils 2.33 or above.
  - Glibc 2.32 or above.
    - CET backports to 2.30/2.31 branches are available.
  - C, C++ and Fortran only.
- Fedora 33 and Ubuntu 20.10 are 2 of CET enabled Linux OSes.
  - It has been done piece by piece. Not all OS pieces have been CET enabled.
    - No performance impact on legacy processors.

Linux Kernel Status

- XSAVES supervisor state may be merged in v5.7.
- Shadow Stack v10 is sent 4/29, may be merged in v5.8.
- Indirect Branch Tracking may be merged after v5.9.
- The current CET kernel patches are available from:
  - https://github.com/yyu168/linux_cet/commits/cet
- The stable CET kernel patches for v5.7 are available from:
  - https://github.com/hjl-tools/linux/tree/hjl/cet/linux-5.7.y

CET Smoke Test

CET smoke test: https://gitlab.com/cet-software/cet-smoke-test

- Check basic CET functionalities on CET enabled Linux OS:
- Quick tests:
  - All tests should run successfully.
- Violation tests:
  - Tests with forced CET violations.
  - All tests should fail with segfault due to CET violation.

CET Smoke Test vs. Kernel Self-test

CET smoke test

- Only run on CET enabled Linux OS.
- Verify if CET is functional.

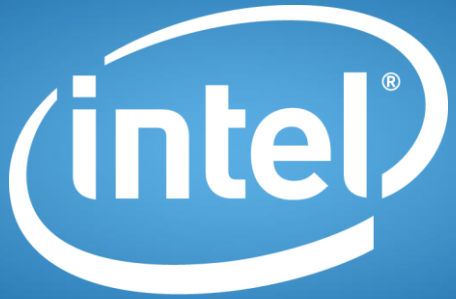
Kernel self-test

- Run on both CET and non-CET Linux OSes.
- Verify that kernel isn't broken when CET is enabled.
- Do not verify if CET is functional.
- Adding one test to verify if CET is functional:
  - Need to run on CET enabled OS.

Call To Action

Enable CET in the rest 5% of Linux OS.

- Other high level languages:
  - Rust
  - Go
  - ...
- Browsers: Chrome, Firefox
  - Javascript
  - Sandbox.
- Java: OpenJDK.



Software

Disclaimers

•Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

•No computer system can be absolutely secure.

•Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

•Intel, the Intel logo and others are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.

Linux Kernel CET Loader

When loading an executable, kernel checks its CET marker and performs CET preparation:

- If the CET marker exists, kernel enables CET features according the CET marker.
  - Kernel allocates shadow stack if SHSTK is enabled.
- Otherwise, kernel disables all CET features.
- For static executables, kernel passes control to the executable.
- For dynamic executables, kernel checks the CET marker on the dynamic loader and passes control to the dynamic loader.
  - On CET enabled OS, the dynamic loader is CET enabled.

Fork, Clone and Exec

- Fork/Clone syscall: The child inherits the parent's CET status:
  - Kernel allocates a new shadow stack for child if SHSTK is enabled.
- Exec syscall: Kernel initializes CET status when loading the executable.

Glibc CET Loader

When control passed from kernel, loader calls arch_prctl (ARCH_CET_STATUS) to check if CET is enabled. When CET is enabled:

- If this is a static executable, calls arch_prctl (ARCH_CET_LOCK) to lock CET, if CET control isn't permissive, and continue.
- Else (a dynamic executable), check the CET marker on all loaded share objects.
  - If any CET marker is missing, call arch_prctl (ARCH_CET_DISABLE) to disable CET.
  - Else calls arch_prctl (ARCH_CET_LOCK) to lock CET if CET control isn't permissive
  - Continue.

dlopen

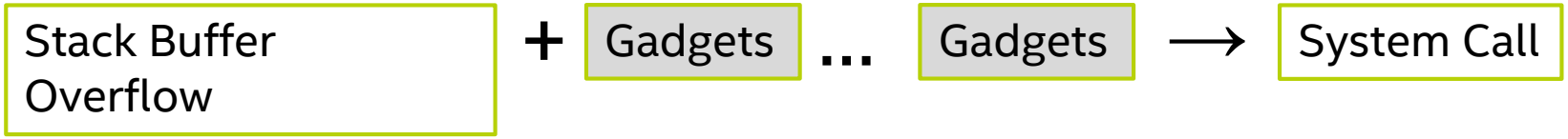
When dlopening a shared object from a CET-enabled process, loader checks the CET marker on all share objects loaded by dlopen:

- If any CET marker is missing:
  - If CET control is permissive, call arch_prctl (ARCH_CET_DISABLE) to disable CET.
  - Else issue an error.

arch_prctl (ARCH_CET_LOCK)

Glibc loader calls arch_prctl (ARCH_CET_LOCK) to lock CET, if CET control isn't permissive, before passing control to main () to prevent disabling CET by accident or malicious attempt.

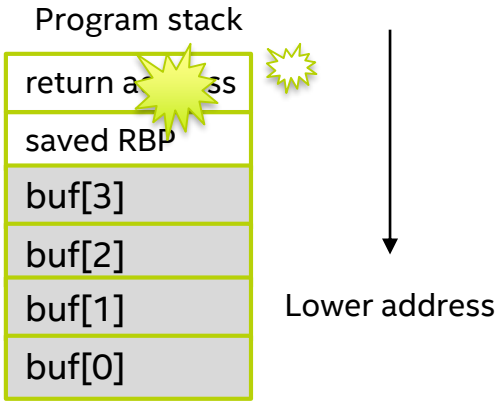
Return/Jump Oriented Programming (ROP) Attacks



No code injection is needed!

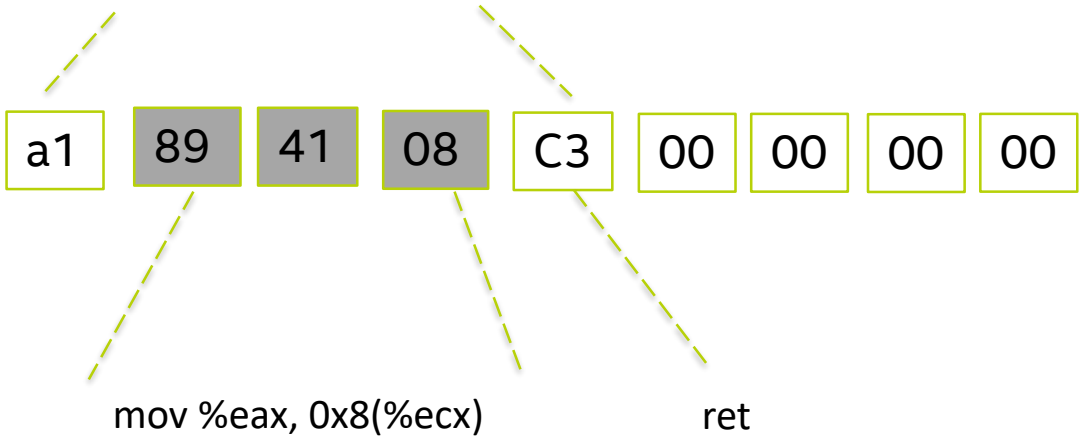
The Stack Buffer Overflow

```
void copy_string(char *input)
{
  char buf[4];
  memcpy(buf, input, strlen(input));
}
```

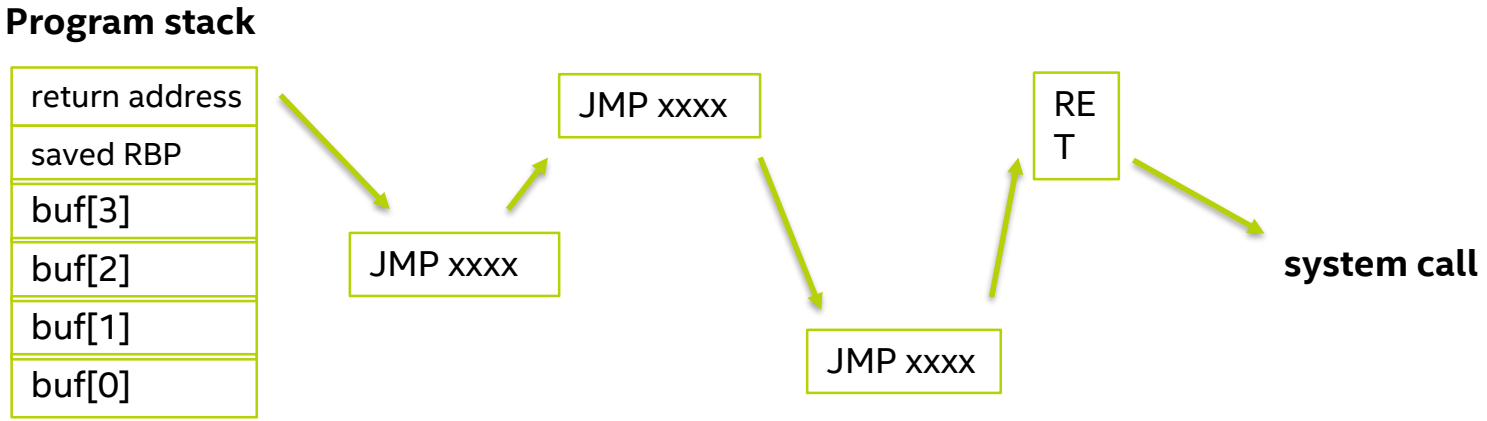


A Code Gadget Example

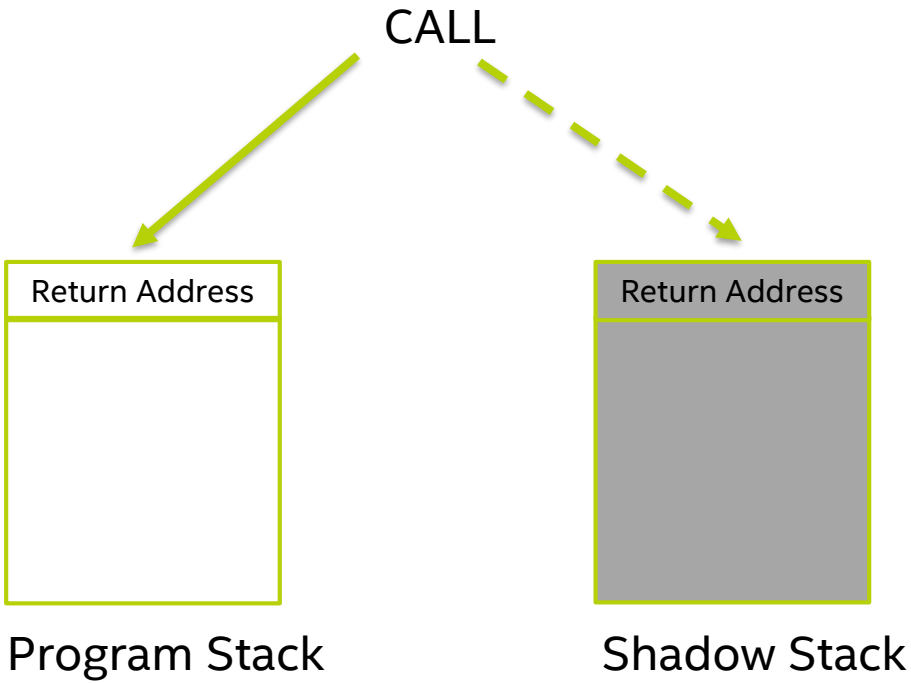
mov 0xc3084189, %eax



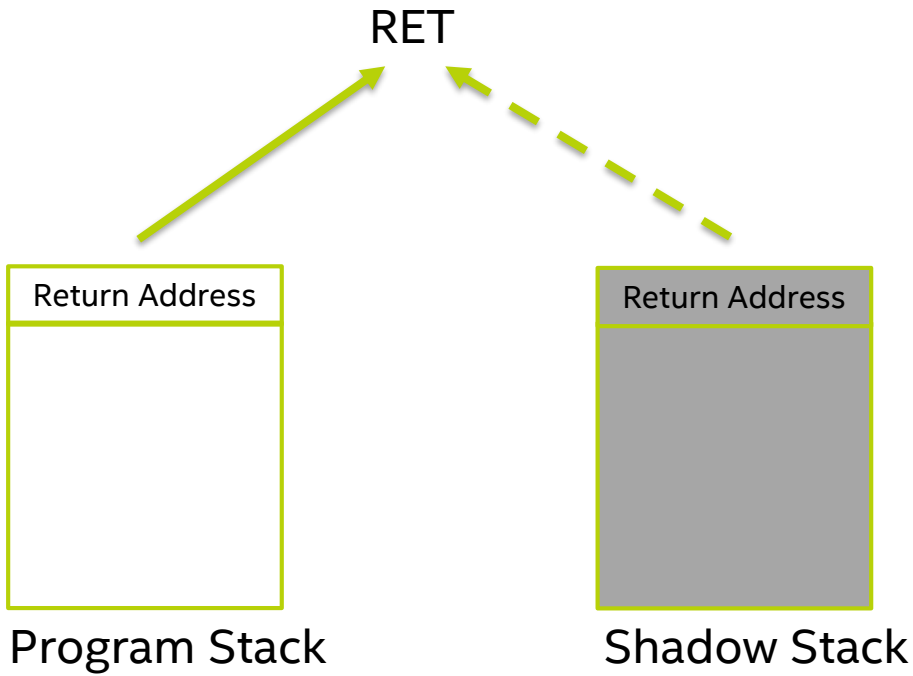
The ROP Attack



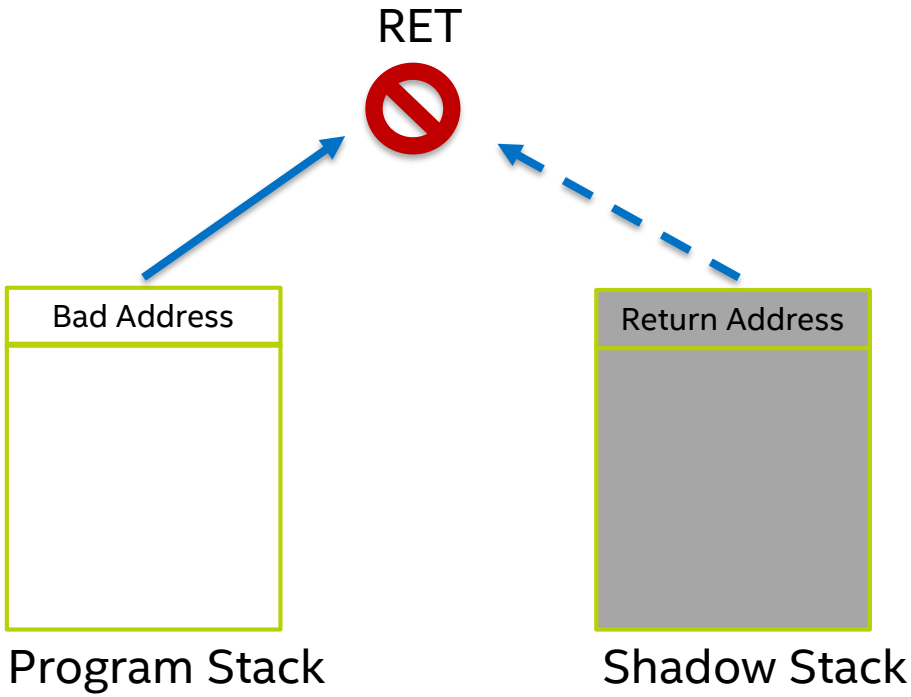
Shadow Stack Concept



Shadow Stack -- Return Address Matching



Shadow Stack Exception



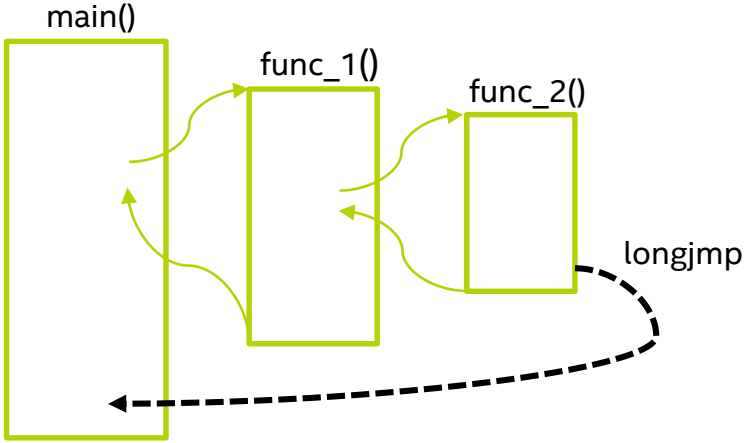
New CET Instructions

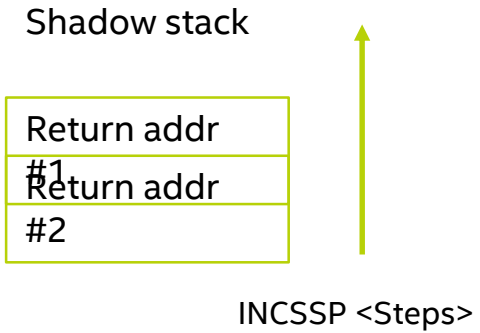
- RDSSP – Read shadow stack pointer
- INCSSP – Shadow stack unwinding

- RSTORSSP, SAVEPREVSSP – Shadow stack context switching
- SETSSBSY, CLRSSBSY – Mark shadow stack in-use

- ENDBR, NoTrack – Indirect branch tracking

Shadow Stack Unwinding





Indirect Branch Tracking (IBT)

```
main() {
     int (*f)();
     f = test;
     f();
}

int test() {
     return
}
```

```
<main>:
ENDBR
:
movq    $0x4004fb, -8(%rbp)
mov     -8(%rbp), %rdx
call    *%rdx
:
retq

<test>:
ENDBR
:
add rax, rbx
:
retq
```