`control-flow-guard`

The tracking issue for this feature is: #68793.

The rustc flag `-Z control-flow-guard` enables the Windows Control Flow Guard (CFG) platform security feature.

CFG is an exploit mitigation designed to enforce control-flow integrity for software running on supported Windows platforms (Windows 8.1 onwards). Specifically, CFG uses runtime checks to validate the target address of every indirect call/jump before allowing the call to complete.

During compilation, the compiler identifies all indirect calls/jumps and adds CFG checks. It also emits metadata containing the relative addresses of all address-taken functions. At runtime, if the binary is run on a CFG-aware operating system, the loader uses the CFG metadata to generate a bitmap of the address space and marks those addresses that contain valid targets. On each indirect call, the inserted check determines whether the target address is marked in this bitmap. If the target is not valid, the process is terminated.

In terms of interoperability:

- Code compiled with CFG enabled can be linked with libraries and object files that are not compiled with CFG. In this case, a CFG-aware linker can identify address-taken functions in the non-CFG libraries.
- Libraries compiled with CFG can linked into non-CFG programs. In this case, the CFG runtime checks in the libraries are not used (i.e. the mitigation is completely disabled).

CFG functionality is completely implemented in the LLVM backend and is supported for X86 (32-bit and 64-bit), ARM, and Aarch64 targets. The rustc flag adds the relevant LLVM module flags to enable the feature. This flag will be ignored for all non-Windows targets.

When to use Control Flow Guard

The primary motivation for enabling CFG in Rust is to enhance security when linking against non-Rust code, especially C/C++ code. To achieve full CFG protection, all indirect calls (including any from Rust code) must have the appropriate CFG checks, as added by this flag. CFG can also improve security for Rust code that uses the `unsafe` keyword.

Another motivation behind CFG is to harden programs against return-oriented programming (ROP) attacks. CFG disallows an attacker from taking advantage of the program's own instructions while redirecting control flow in unexpected ways.

Overhead of Control Flow Guard

The CFG checks and metadata can potentially increase binary size and runtime overhead. The magnitude of any increase depends on the number and frequency of indirect calls. For example, enabling CFG for the Rust standard library increases binary size by approximately 0.14%. Enabling CFG in the SPEC CPU 2017 Integer Speed benchmark suite (compiled with Clang/LLVM) incurs approximate runtime overheads of between 0% and 8%, with a geometric mean of 2.9%.

Testing Control Flow Guard

The rustc flag `-Z control-flow-guard=nochecks` instructs LLVM to emit the list of valid call targets without inserting runtime checks. This flag should only be used for testing purposes as it does not provide security enforcement.

Control Flow Guard in libraries

It is strongly recommended to also enable CFG checks for all linked libraries, including the standard library.

To enable CFG in the standard library, use the cargo `-Z build-std` functionality to recompile the standard library with the same configuration options as the main program.

For example:

```
rustup toolchain install --force nightly
rustup component add rust-src
SET RUSTFLAGS=-Z control-flow-guard
cargo +nightly build -Z build-std --target x86_64-pc-windows-msvc
```

```
rustup toolchain install --force nightly
rustup component add rust-src
$Env:RUSTFLAGS = "-Z control-flow-guard"
cargo +nightly build -Z build-std --target x86_64-pc-windows-msvc
```

Alternatively, if you are building the standard library from source, you can set `control-flow-guard = true` in the config.toml file.