# Contributing

Before we can accept a pull request from you, you'll need to sign a [Contributor License Agreement (CLA)](https://cla.microsoft.com). It is an automated process and you only need to do it once.

To enable us to quickly review and accept your pull requests, always create one pull request per issue and link the issue in the pull request. Never merge multiple requests in one unless they have the same root cause. Be sure to follow our Coding Guidelines and keep code changes as small as possible. Avoid pure formatting changes to code that has not been modified otherwise. Pull requests should contain tests whenever possible.

# Branching

The master branch contains current development. While CI should ensure that master always builds, it is still considered pre-release code. Release checkpoints will be put into stable branches for maintenance.

To contribute, fork the repository and create a branch in your fork for your work. Please keep branch names short and descriptive. Please direct PRs into the upstream master branch.

## Build and run from source

### Environment

\* Linux OS (dev team is using Ubuntu 18.04)

\* Development can be done on Windows Subsystem for Linux (WSL), but ProcDump cannot be executed in that environment

\* git

\* GDB

\* GCC (or other C compiler)

\* GNU Make

## Building from source

\* Clone the repo

\* Run `make` from the project root

\* The procdump executable will be placed into the `bin` directory

```sh

git clone https://github.com/microsoft/ProcDump-for-Linux

cd ProcDump-for-Linux

make

```

## Testing

\* There are a multitude of tests included in the `tests` directory of the repository.

\* Add new tests corresponding to your change, if applicable. Include tests when adding new features. When fixing bugs, start with adding a test that highlights how the current behavior is broken.

\* Make sure that the tests are all passing, including your new tests.

## Creating integration tests

The integration tests run using the local procdump built from source. Individual test cases are written as bash scripts and need to be inside `/tests/integration/scenarios/` in order to be called automatically by `run.sh`.

Test scripts will return `0` when they succeed and `1` when they fail.

Most of the tests are written using [stress-ng](https://wiki.ubuntu.com/Kernel/References/stress-ng "stress-ng manual"), but you can write your own code to simulate the scenario you require.

After writing a new test, run the `run.sh` script and verify that no tests fail.

## Pull Requests

\* Always tag a work item or issue with a pull request.

\* Limit pull requests to as few issues as possible, preferably 1 per PR

## Coding Guidelines

## Indentation

We welcome all whitespace characters, but prefer space expanded tabs.

## Names

\* Do not use `typedef`, we like to know what is a struct and what is a type

\* Use PascalCase for:

\* `struct` names

\* `enum` names

\* `function` names

\* Use camelCase for:

\* `local variable` names

\* `enum` names should start with a captol `E`, e.g., `enum ECoreDumpType`

\* Global variables should be prefixed with `g\_`, e.g., `struct ProcDumpConfiguration g\_Config;`

\* `struct Handle`s that contain a `struct Event` should have variable names prefixed by `evt`, e.g., `struct Handle evtIsQuit;`

\* `struct Handle`s that contain a `sem\_t` should have variable names prefixed by `sem`, e.g., `struct Handle semDumpSlot;`

\* Please use whole words when possible

## Style

\* Curly brackets, `{ }`, should go on the next line after whatever necessitates them

\* For structs, put on same line

\* Put a space before the open paren, `(`, with `for`, `while`, `if`, and `switch` statements

\* No space after function names and before parameter lists

\* The `\*` for a pointer goes next to the variable name, e.g., `char \*variable`

\* Declare 1 variable at a time

\* Declare all local variables at the start of a function

\* Either initialize upon declaration, or initialize before logic

\* The exception is `for` loop iterators

\* Wrap header (`.h`) files with:

```c

#ifndef HEADER\_FILE\_NAME\_H

#define HEADER\_FILE\_NAME\_H

//...

#endif // HEADER\_FILE\_NAME\_H

```

## Trace and Error Handling

For system calls and other "failable" function calls, please make use of the `Trace` macro and the logging methods, like so:

```c

int rc = 0;

if ((rc = FailableFunction(...)) != 0)

{

Log(error, INTERNAL\_ERROR);

Trace("WriteCoreDump: failed pthread\_setcanceltype.");

exit(-1);

}

```

## Example of style

```c

struct Baz {

int Foobar;

}

int main(int argc, char \*argv[])

{

int foo = 0;

int bar = 1;

char[64] str = "This is a string";

struct Baz baz = { 10 };

while (foo < 10)

{

foo++;

}

for (int i = 0; i < foo; i++)

{

printf(str);

baz.Foobar--;

}

printf("baz.Foobar is %d", baz.Foobar);

return bar - 1;

}

```