Todo/Notes

**General**

* Todo list.
  + Rewrite this list. (After outlining a ‘definition of done’.)
  + Consider porting to Google Code issue tracker.
* Documentation.
  + Improved and expanded documentation. Add notes, cautions, symbols, cross-refs, etc.
  + Threading guarantees.
  + Exception guarantees.
  + Document preconditions, postconditions, and invariants.
  + Better annotations (warnings, notes, etc).
  + Note where handles are inheritable.
  + Note what access control handles are granted.
  + Improved commenting of components.
  + Update doc gen tools and improve doc gen process.
* Tests.
  + Write a specially crafted process to use as a sample ‘target’ for unit tests and examples. This way known values can be checked for cases where they are typically unknown, even for our own process.
  + Clean up tests.
  + Extra sanity checks to ensure not only that functions return without exception, but also that the returned data is valid. (e.g. GetName in ImpThunk.)
  + More tests for cases that should throw (BOOST\_CHECK\_THROW).
  + Improved test coverage of all APIs (including operator overloads, copy/move constructors, etc).
  + Split up test cases further.
  + Rewrite tests to be less ‘hackish’ and ‘ugly’.
  + Tests for examples.
* Examples.
  + Clean up existing example code.
  + Demonstrate exception handling.
  + Demonstrate multi-threading.
  + Basic memory hacking apps for each component.
* Re-add all components from v1.0.0.
  + FindPattern
  + PeLib
  + ManualMap
  + Scanner
  + Patcher
  + Disassembler (?)
* Review current code for compliance to relevant coding standards/guidelines (Google, Boost, LLVM, etc).
  + Ensure to only conform to standards that apply. Some rules do not apply outside a particular organization (due to history, compatibility, platform, etc) or scope (particularly low level, extreme high performance, etc).
* Optimize all components, both in terms of interface and implementation.
* Minimize use of Boost and other 3rd party libraries where possible.
* Look for places where cleanup that could potentially fail is done before throwing an exception, and consider taking the exception information from cleanup failure (if it occurs) and attaching it to the exception we’re about to throw. Currently we simply ignore the failure and only report the error that is detected first (which is bad, because it means bugs can be missed).
  + Even if this is not done, it must be documented which cleanup operations can potentially fail without being reported.
  + Examples: CreateAndInject environment modification, Read/Write memory protection modification, pretty much anywhere CloseHandle is called, some uses of Boost.ScopeExit, etc.
  + What about resource cleanup failures in destructors? These should at the very least be logged somewhere.

**New Modules**

* Remote memory ‘pool’ to avoid allocating entire pages for only a few bytes of data.
* Python bindings.
  + Important! Ensure -fno-strict-aliasing is used under GCC as it seems Boost.Python has aliasing violations which cause spurious segfaults and other issues.
* Basic base hook.
* Helper service to run HadesMem tools as ‘SYSTEM’ for when manipulating certain protected/critical processes (running in separate desktops, sessions, etc.).
* Debugging APIs (sw bp, hw bp, conditional bp, single stepping, stack trace, etc).
  + Investigate the debugger engine API (<http://goo.gl/ZV6K7>).
* Thread information and enumeration.
* Remote code ‘emulator’.
* Custom GetModuleHandle, GetProcAddress, etc.

**ManualMap**

* Exception handling support under x86 SafeSEH and x64.
* Improved TLS support.
* CLR hosting support.

**FindPattern**

* Pattern generator.
* ‘Multi-pass’ support (e.g. search for pattern, apply for manipulators, use as starting point for second search).
* Arbitrary region support.

**PeLib**

* Investigate use of virtual functions for file vs memory access (RvaToVa).
  + Alternatively, investigate use of templates, which may ‘merge’ better with x86/x64 cross compatibility.
  + Note: May cause problems when copying ‘PeFile’ type.
* Extra sanity checking in all components.
  + E.g. Check NumberOfRvaAndSizes in NtHeaders before attempting to retrieve a data dir.
* Cache base pointers etc rather than retrieving it manually in every getter/setter. Slightly less ‘robust’, but due to the typically ‘read-only’ nature of the data this is the expected behaviour in all known cases anyway.
* Support more of the PE file format.
  + Overlay data.
  + Resource directory.
  + Exception directory.
  + Relocation directory.
  + Security directory.
  + Debug directory.
  + Load config directory.
  + Delay import directory.
  + Bound import directory.
  + IAT (as opposed to Import) directory.
  + CLR runtime directory support.
* Full support for writing back to PE file, including automatically performing adjustments where required to fit in new data or remove unnecessary space.
* Improve export forwarding code to detect and handle forward-by-ordinal explicitly rather than forcing the user to detect it and do string manipulation and conversion.
* Helper functions such as FindExport, FindImport, HasDataDir, GetArchitecture, IsDotNet, GetPDB, etc.
* Test against pathological cases such as Corkami tests.

**Disassembler**

* Decode calls/jumps to function names if possible.
* NOP/UnNOP support.
* ASM searching API with ‘wildcards’. (MetaASM?)
* String based assembler with x64 support.
* Disassemble function API.

**Scanner**

* Rewrite to be more reliable and robust.
* Refactor to reduce code duplication.
* Use a file view with a small memory cache rather than consuming large amounts of RAM.
* Multi-threaded scanning options.
* Wildcard support for vector/string scanning.
* Regex support for string scanning.
* Memory protection filters (read, write, exec).
* Memory type filters (private, mapped, image).
* Support pausing target while scanning.
* Support injected scanning.
* Configurable scan buffer size.
* Pointer scanner.
* Unknown value scan.
* Progressive scan filtering based on either value or criteria.
* Scan history and undo.
* Support case insensitive string scanning.
* Binary scanning.
* Custom scanning via user-supplied predicate.
* Improved floating point support (configurable or ‘smart’ epsilon).
* Group search support.