Todo

**General**

* ‘Finish’ and stabilize public API.
  + Use semantic versioning. (See <http://semver.org/>)
  + Remove ‘unnecessary’ APIs for now (YAGNI) and focus on a proper design and implementation of the more critical ones.
  + Remove unnecessary 3rd party dependencies from public interface where possible (e.g. Take a std::wstring as a path then convert to a boost::filesystem::path internally to avoid Boost updates breaking the HadesMem ABI).
  + Move ‘internal’ code to ‘detail’ folder/namespace.
* Documentation.
  + QuickBook generated.
  + Write full in-source docs simultaneously.
  + Write threading docs simultaneously.
  + Write exception guarantee docs simultaneously.
* Unit tests.
  + Pass application verifier.
  + Leak testing.
  + In tests such as PeLib etc ensure at least one module is processed per test, in case the validity check function is the one with a bug.
  + 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.)
  + Expand tests to be more thorough and use Boost.Test checks such as BOOST\_CHECK\_NO\_THROW, BOOST\_CHECK\_THROW, BOOST\_CHECK\_EQUAL\_COLLECTION, etc.
  + Extra tests on library usage that should fail to compile (because of static assertions, type traits, etc). Needed to ensure library misuse is not ‘possible’.
* Examples.
  + All components.
  + Cleanup and in-source documentation.
  + Pass application verifier.
  + Leak testing.
* Improve quality of existing code.
  + Improve exception-safety guarantees. (Rollback support etc)
  + Improve thread-safety guarantees and threading model.
  + Throw custom error codes as part of exceptions. Use a different tag type.
  + Investigate Boost.Exception’s support for multiple exception info objects which share the same tag type, and how to ‘extract’ them at the catch site. Potentially rewrite some exceptions which use multiple ‘ErrorCodeWin’ objects to use unique tags instead.
  + Proper localization and internationalization support/framework.
  + Consider moving from UTF-16 internally to UTF-8, and only using UTF-16 at API boundaries. If implemented, all files and other IO should be in UTF-8.
  + Template concept checking and improved error messages.
  + Compile time checks to ensure MSVC, GCC, Boost, etc meet the minimum requirements.
  + Assertions where appropriate (for ‘impossible’ cases).
  + Improved error reporting via exceptions.
  + Support move semantics where appropriate.
  + Support swapping where appropriate (and use to simplify operator= if possible).
  + Support perfect forwarding where appropriate.
  + Style cleanup (e.g. LLVM).
  + Const support for iterators (template iterators on ValueT, add cbegin/cend to parent).
  + Fix conversion to bool in classes (e.g. EnsureCleanup) to use an unspecified bool type and prevent implicit conversion to int, and also to remove the reliance on HandleType’s implicit conversion to bool, which we cannot guarantee.
  + Const correctness.
  + Clean up iterators to be more lightweight, and shift most of the processing to the parent object.
* Improved binary generation and distribution.
  + Binaries with update server.
  + Automatic binary versioning using SVN revision.
  + Automatic build bot.
  + Digital signature (for build bot, WER, etc).
* Improve API.
  + Clean up APIs returning pairs/tuples/etc to return objects with named fields and/or getters.
  + Investigate support for optional error reporting via error codes to avoid littering code with try-catch blocks in certain scenarios.
  + Improve API genericity.
    - Instead of taking or returning a vector<T>, instead use a C<T> where ‘C’ is a template parameter representing an arbitrary container.
    - Use ‘ranges’ over direct containers where possible.
* Reduce compile time.
  + Reduce include dependencies.
    - Pointer-to-impl idiom.
  + PCH support using Boost.Build.
  + Decouple components if possible.
* Todo list.
  + Consider porting to Google Code issue tracker.
* Maintain a changelog.
* Performance improvements in potential bottlenecks (e.g. Iterators, Scanner, PeLib, FindPattern, etc).

**New Modules**

* Cheat Engine ‘replacement’.
* Helper service to run HadesMem tools as ‘SYSTEM’ for when manipulation certain protected/critical processes (running in separate desktops, sessions, etc.).
  + Ensure service is secure from abuse.
* Debugging.
  + Software breakpoint.
  + Hardware breakpoint (including memory read/write detection).
  + Conditional breakpoints.
  + Single stepping.
  + Stack trace.
* Anti-anti-cheat.
* Thread/process information and enumeration.
* Manual mapper.
  + Windows PE loader reversing.
  + TLS support.
  + Exception handling support. (Safe, rather than catch-all.)
  + No ‘external’ (read: MMHelper) dependencies.
  + Extensive sanity checking on PE file.
  + Manually map dependencies and ‘link’.
* Remote code ‘emulator’.
* Custom GetModuleHandle, GetProcAddress, etc.
* C++ WinAPI wrapper.

**Process**

* Constructors that take window or process name should throw if multiple matches are found. In this case, the PID should be used.

**MemoryMgr**

* Clean up memory reading/writing API to reduce use of template metaprogramming in public interface.
* Improve ‘safety’ of remote function caller via EH to minimize risk of crashing the target.
* Improve genericity of parameter passing in remote function caller.
* Memory reading via expression templates.
* Improved type genericity via type decay to ensure templates function correctly in the face of const, volatile, references, etc. (Perfect forwarding?)
* Fix type traits in Read/Write APIs to support std::vector< const T> and other ‘mutations’ that are not currently handled.
* Support floating point return values and parameters in remote function caller.
* Support 64-bit return values under x86 in remote function caller (EAX:EDX).
* Support non-MSVC compilers in remote function caller (e.g. in calling convention specification).

**Module**

* Remote GetModuleHandle implementation with same path semantics etc.

**Injector**

* .NET injection.
  + Without DLL dependency if possible.
* Cross-section injection.
* IAT injection.
* Varargs/generic export calling.
* Consider NT API based injection (LdrLoadDll).
* Add ‘FreeDll’ API.
* Get address of Kernel32!LoadLibrary ‘manually’ rather than using a local GetProcAddress and pointer arithmetic.
  + Whilst this works in all normal cases, it doesn’t work when the target has shims enabled which hook LoadLibrary.

**Patcher**

* VEH hooking (both INT3 and DR).
* Transactional hooking.
* Improved relative instruction rebuilding (including conditionals). x64 has far more IP relative instructions than x86.
* Freeze target when hooking (except calling thread if applicable – e.g. in injected code).
* When hooking on x64 try to find a free memory block for the trampoline that is within RIP-relative range of the detour. Only if one cannot be acquired should we fall back to a system-provided address and an absolute jump.
* Uncopyable, so make moveable.
* VMT hooking.
* IAT/EAT hooking.
* Explicitly support hook chains (and write test).
* Use relative jumps where possible (detect delta at runtime).
* Detect cases where hooking may overflow past the end of a function, and fail. (Provide policy or flag to allow overriding this behaviour.) Examples may be instructions such as ‘int 3’, ‘ret’, ‘jmp’, etc.

**FindPattern**

* Pattern generator.
* Multi-threaded scanning.
* ‘Start’ attribute support.
* Data/Rdata scanning support (for ‘start’ attribute and constant scans).
* Dump results to file.
* Custom file format for specifying patterns.
* Object or stream-based implementation that allows operation chaining.

**PeLib**

* Support for working on x86 PE files from x64 and vice versa.
* Finish implementing matching ‘setters’ for existing ‘getters’.
* Investigate use of virtual functions for file vs memory access (RvaToVa).
* 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.
* Consistent API for GetBase. (e.g. PVOID vs PBYTE vs T\*)
* 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.

**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.