Todo

**General**

* Rewrite this list. (After outlining a ‘definition of done’.)
* Todo list.
  + 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 and postconditions.
  + Better annotations (warnings, notes, etc).
  + Note where handles are inheritable.
  + Note what access control handles are granted.
* Unit 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.
  + Split tests even further. E.g. Read -> ReadPod, ReadList, ReadString, etc.
  + 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’.
  + Test copying, assignment, etc.
  + Add concept checks in unit tests for constructability, moveability, destructibility, etc.
* Improve quality of existing code.
  + Convert standalone iterators to be created by factory class (process\_iterator -> Processes::begin/end, etc).
  + Inspect code for aliasing violations.
  + Use noexcept where appropriate, and check existing cases where noexcept is used.
  + Explicitly qualify calls to functions in the global namespace (Windows API etc) for clarity.
  + Use ‘auto’ by default where appropriate.
  + Take advantage of defaulted and deleted functions (macro wrapped).
  + Improve exception-safety guarantees. (Rollback support etc)
  + Improve thread-safety guarantees and threading model.
  + Template concept checking and improved error messages.
  + Assertions where appropriate (for ‘impossible’ or rare cases).
    - Especially check for things like integer overflows that would otherwise be hard to debug.
  + Use ‘ranges’ over direct containers where possible.
  + Investigate nested exceptions (std::throw\_with\_nested etc.).
* Stack trace in exception object.
* Performance improvements in potential bottlenecks (e.g. Iterators, Scanner, PeLib, FindPattern, etc).

**New Modules**

* Process enumeration.
* 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.
* Cheat Engine ‘replacement’.
* Helper service to run HadesMem tools as ‘SYSTEM’ for when manipulating certain protected/critical processes (running in separate desktops, sessions, etc.).
* Debugging.
  + Software breakpoint.
  + Hardware breakpoint (including memory read/write detection).
  + Conditional breakpoints.
  + Single stepping.
  + Stack trace.
* Anti-anti-cheat.
* Thread/process information and enumeration.
* Remote code ‘emulator’.
* Custom GetModuleHandle, GetProcAddress, etc.
* Hookshark-style tool.
* Manually mapped module detection.

**MemoryMgr**

* Use Boost.FunctionTypes in remote function caller to detect calling convention. (Also use TMP to detect number of args and their types etc).
* 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.
* Support floating point parameters and return values in remote function caller.
* Support 64-bit parameters under x86 in remote function caller.
* Support non-MSVC compilers in remote function caller (e.g. in calling convention specification).

**ManualMap**

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

**Injector**

* .NET injection.
  + Without DLL dependency if possible.
* Cross-section injection.
* IAT injection.
* 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-pass’ support (e.g. search for pattern, apply for manipulators, use as starting point for second search).
* Arbitrary region support.

**PeLib**

* Support for working on x86 PE files from x64 and vice versa.
* 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.