UNCLASSIFIED//FOUO

(U//FOUO) In-memory Code Execution Specification

(U//FOUO) Specification 002: version 3 final

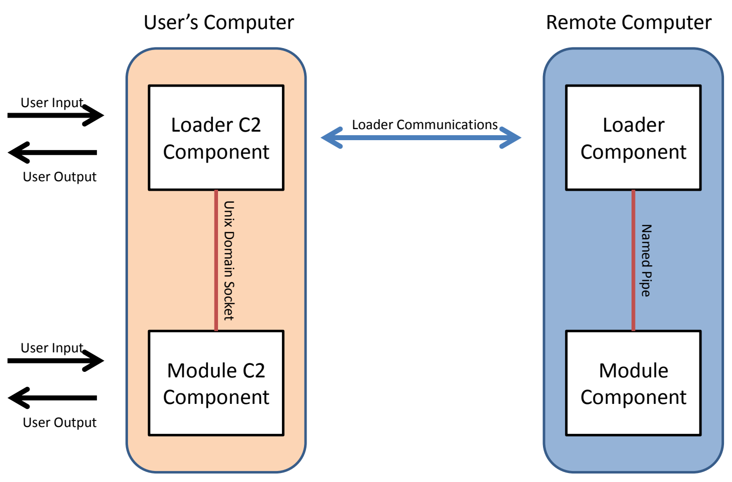
UNCLASSIFIED//FOUO

(U//FOUO) Contents

1. (U//FOUO) Overview ... 3
2. (U//FOUO) Loading and Invocation ... 5
3. (U//FOUO) Exported Function ... 6
   3.1. (U//FOUO) Definition ... 6
   3.2. (U//FOUO) Arguments ... 7
      3.2.1. (U//FOUO) Structure Elements ... 7
4. (U//FOUO) Metadata ... 9
   4.1. (U//FOUO) Metadata Elements and Attributes ... 10
5. (U//FOUO) Feature Sets ... 11
   5.1. (U//FOUO) Fire ... 11
   5.2. (U//FOUO) Fire and Forget ... 11
   5.3. (U//FOUO) Fire and Collect ... 11
      5.3.1. (U//FOUO) Output Pipe ... 12
   5.4. (U//FOUO) Fire and Interact ... 12
      5.4.1. (U//FOUO) Input and Output Pipe ... 12
6. (U//FOUO) Communicating Status ... 13
   6.1. (U//FOUO) Loader Responsibilities ... 13
   6.2. (U//FOUO) Module Responsibilities ... 13
7. (U//FOUO) Return Values ... 14
8. (U//FOUO) Load and Execute Loop ... 15
9. (U//FOUO) Module Caching ... 15
10. (U//FOUO) Structured Exception Handling ... 15
11. (U//FOUO) Vectorized Exception Handling ... 15
12. (U//FOUO) Pipe Opening and Closing Order ... 15
13. (U//FOUO) Dependency Resolution and Ordinals ... 16
14. (U//FOUO) Hooking From Multiple Directions ... 16
15. (U//FOUO) Behaviors to Avoid ... 16

Appendix A. (U//FOUO) Version History ... 17
Appendix B. (U//FOUO) Metadata XML DTD ... 19
Appendix C. (U//FOUO) Metadata XML XSD ... 20
Appendix D. (U//FOUO) Future Work ... 23

1. (U//FOUO) Overview



(U//FOUO) Figure 1: An overview of ICE components and their execution location

(U//FOUO) In-memory Code Execution (ICE) modules are relocatable portable executable files (normally DLL files) that expose a function exported by ordinal. These modules are designed to be used by an ICE-aware loader that loads the module file and creates a thread that calls the exported function without the module code being written to disk.

(U//FOUO) Four major categories of behavior ("feature sets") are defined:

- **Fire** – the module is loaded, its exported function is executed, and the module is unloaded upon function return. This feature set is best suited for modules which perform small controllable changes or checks without the need to continue running or return status other than an exit code.

- **Fire and Forget** – the module is loaded, its exported function is executed, and the module is unloaded when it decides to end execution (which may last beyond the return from the exported function). This feature set is best suited for modules which may continue running for some period of time but will not need to communicate or interact with the loader for that duration. Modules are not truly "forgotten" by the loader, some tracking state as defined below is maintained, this name instead refers to the user's ability to concentrate elsewhere.

- **Fire and Collect** – the module is loaded, its exported function is executed, and the module is unloaded when it decides to end execution (which may last beyond the return from the exported function) or when directed by the loader. The module may provide output to the loader for passage to the user. This feature set is best suited for modules which need to provide some sort of output back to the user in a consistent and secure manner.

- **Fire and Interact** – the module is loaded, its exported function is executed, and the module is unloaded when it decides to end execution (which may last beyond the return from the exported function) or when directed by the loader. The module may provide output to the loader for passage to the user and may receive input from same. This feature set is best suited for modules needing the full gamut of bi-directional communication with the user or processes on the user's workstation.

(U//FOUO) The ICE specification is primarily designed for in-memory execution within a single process though efforts have been made to accommodate code injection into remote processes. Due to the complexity necessary to accommodate the primary goals of this specification some "nice to have" features which would make remote injection simpler for the user have been deferred to future versions.

(U//FOUO) As shown in the table below, each feature set is a subset and superset of other feature sets. These relationships are expressed through which parameters a feature set may utilize. A more detailed discussion of the parameters is in section 3.2 below.

-	Fire	Fire and Forget	Fire and Collect	Fire and Interact
`version`	X	X	X	X
`hModule`	X	X	X	X
`cmdline`	X	X	X	X
`behavior`	X	X	X	X
`moduleThread`		X	X	X
`moduleQuit`			X	X
`pipeName`			X	X

(U//FOUO) Figure 2: Feature Set parameter support

2. (U//FOUO) Loading and Invocation

2.1. (U//FOUO) For Fire and Collect or Fire and Interact modules, the loader's Command and Control (C2) component connects to a Unix domain socket (UDS) identified by a user-supplied prefix.

2.1.1. (U//FOUO) The UDS is created by a module-author supplied C2 handler and the contents of the Fire and Collect or Fire and Interact pipe is passed from the loader's C2 component to the module handler for processing and user interaction. The module handler is invoked by the user separately from the loader C2 component and, at a minimum, takes an argument of a user-supplied UDS name.

2.1.2. (U//FOUO) The loader should establish a connection to the module handler via the UDS before attempting to load the module on a remote machine

2.1.3. (U//FOUO) If either the module handler or the loader's C2 component detect that the other process has closed their end of the pipe then they must notify the user, close their end of the pipe, and clean up the UDS from the user's system.

2.2. (U//FOUO) Loader maps the module into memory, performing fixups as appropriate, and recursively loads any dependencies.

2.2.1. (U//FOUO) The loader should apply the default memory page permissions for a module's sections (i.e., as would be set by `LoadLibrary`). This prevents a single large chunk of `PAGE_EXECUTE_READWRITE` memory which could easily be found by memory forensics.

2.3. (U//FOUO) Loader calls `DllMain` and then the exported `MemoryLoad` function as described below.

2.3.1. (U//FOUO) The loader must not call `DllMain` or `MemoryLoad` from a main thread to avoid a deadlock situation caused by a buggy module.

2.3.2. (U//FOUO) If `DllMain` or the exported function is not present, then the loader will unload the module and zero the memory it occupied.

2.3.3. (U//FOUO) If `DllMain` or the exported function returns a negative, other than as allowed in section 7 below, then the loader will stop interaction with the module. This is due to the uncertain nature of the module's state in erroneous situations.

2.3.4. (U//FOUO) The thread the loader uses to call `DllMain` and `MemoryLoad` should be returned promptly. For Fire modules this means after the module's execution is completed. For Fire and Forget and higher modules this means that `MemoryLoad` should use this thread for initialization only and spawn additional threads as needed to perform the module's function.

2.3.5. (U//FOUO) Modules should note that the thread the loader uses to call `DllMain` and the thread the loader uses to call `MemoryLoad` may not be the same thread. Thread-local storage is therefore not guaranteed to persist between those calls.

2.4. (U//FOUO) Upon `MemoryLoad`'s return, the module and any dependencies are unloaded and the memory it occupied is zeroed. This behavior can be overridden if desired for Fire and Forget or higher modules.

2.4.1. (U//FOUO) Modules should react to subsequent invocations of `DllMain` as per the MSDN specification.

2.4.2. (U//FOUO) Loaders must call a module's `DllMain` with reason code `DLL_PROCESS_DETACH` prior to unloading that module.

2.4.3. (U//FOUO) Loaders should expect that Fire and Forget or higher modules may not end prior to the loader ending execution. This is explicitly allowed to permit a module to upgrade a loader via a separate connection. This is also allowed to permit modules which find themselves unexpectedly unable to be unloaded to prevent the loader from ever attempting to unload the module.

3. (U//FOUO) Exported Function

3.1. (U//FOUO) Definition

(U//FOUO) A module exposes an exported function with the prototype

```
DWORD WINAPI MemoryLoad(__in LPVOID run_arg_struct);
```

(U//FOUO) Loaders will not use the name of the function when resolving exports, instead invoking the function by ordinal only. Modules may include a function name if desired, this name should be meaningless and innocuous (i.e., not "FaF" or "MemoryLoad"). Additional exported functions may be used by the module to disguise the true entry point.

(U//FOUO) An example ICE module `.def` file. Note that the exact ordinal to call should vary and must be included in the metadata file (specified below):

```
LIBRARY module
EXPORTS
     MemoryLoad            @7 NONAME
```

(U//FOUO) For the Fire feature set `MemoryLoad` is the entirety of module execution. For Fire and Forget, Fire and Collect, and Fire and Interact the `MemoryLoad` function is to be used as an initializer. Actual functionality must be performed after the return from `MemoryLoad`. In particular, the pipe (described in section 5 below) is not guaranteed to have data until after `MemoryLoad` returns and must continue to be available until the module or loader quit.

3.2. (U//FOUO) Arguments

(U//FOUO) The ICE specification provides the ability to run a module with arguments as if it was run from the command line. The module's `MemoryLoad` function is called and the `run_arg_struct` parameter points to a `MODULE_REMOTE_ARGS` structure as defined below:

```
// Ensure proper structure member alignment
#pragma pack(push)
#pragma pack(1)

#define MODULE_ARGS_CMDLINE_LEN (MAX_PATH * 4)
#define MODULE_ARGS_PIPENAME_LEN MAX_PATH

typedef struct _MODULE_REMOTE_ARGS {
  DWORD version;   // Current version is 3
  LPVOID hModule;
  wchar_t cmdline[MODULE_ARGS_CMDLINE_LEN];
  DWORD behavior;
  LPHANDLE moduleThread;
  HANDLE moduleQuit;
  wchar_t pipeName[MODULE_ARGS_PIPENAME_LEN];
} MODULE_REMOTE_ARGS, *PMODULE_REMOTE_ARGS;

#pragma pack(pop)
```

(U//FOUO) The `MODULE_REMOTE_ARGS` structure is designed for future extensibility by including a version number, reported via the structure's `version` field. The version defined by this specification is the same as the version of this specification, not including the revision number. Attempts will be made to maintain backward and forward compatibility in future versions of the ICE specification but no specific guarantees are made.

(U//FOUO) The `MODULE_REMOTE_ARGS` structure will be allocated by the loader and is zeroed and freed by the loader once the module's ordinal function returns. The module should copy any values it wishes to retain beyond that point.

(U//FOUO) Structure elements not required by the requested behavior (e.g., `pipeName`, `moduleQuit`, and `moduleThread` are not required when `behavior` requests Fire) will be initialized to `NULL`.

3.2.1. (U//FOUO) Structure Elements

(U//FOUO) **`version`** – The version of the ICE specification used by the loader for this invocation. The version defined by this specification is the version number on the title page of this specification. See also the version history at the end of this document. If a loader supporting `version M` executes a module supporting `version N` and `M > N` then the loader should use a `version` value of `N` for the invocation. A module should verify that this version parameter is an exact match for its supported `version` and return the appropriate error otherwise (see section 7 below for more detail).

(U//FOUO) **`hModule`** – A pointer to the beginning of the module in memory.

(U//FOUO) **cmdline** – A null-terminated string defining the user provided arguments to the function, the effective equivalent of argv command line arguments to an executable. Note that a placeholder value will be inserted at the beginning of this string to simulate argv[0]. An example: for user supplied arguments "-a -b 8 -c" cmdline will be passed to the module as "0 -a -b 8 -c" where "0" is the placeholder value. The placeholder value is not fixed and has no meaning and will be separated from the other arguments by a space character.

(U//FOUO) **behavior** – A bitmask specifying which feature set this invocation is requesting of the module, the 30 high bits are reserved and the 2 least significant bits are feature set bits. The two least significant bits are defined to mean:

```
#define ICE_FIRE              0x0
#define ICE_FIRE_AND_FORGET   0x1
#define ICE_FIRE_AND_COLLECT  0x2
#define ICE_FIRE_AND_INTERACT 0x3
```

(U//FOUO) **moduleThread** – A pointer to a handle allocated by the loader and initialized to NULL. The module must set the handle to a thread handle if the module desires to continue running after its ordinal function returns. For more information see section 6, Communicating Status, below. Note that this is initialized to NULL vice INVALID_HANDLE_VALUE to avoid a subtle bug with the Windows API call WaitForSingleObject. The loader is responsible for closing *moduleThread when appropriate.

(U//FOUO) **moduleQuit** – A Windows Event handle created by the loader. The loader will signal this event if the module should stop running due to some abnormal circumstances (e.g., connectivity loss, user command). If moduleQuit is signaled a module should quit immediately, there is no guarantee that the pipe is in a usable state.

(U//FOUO) When not required by the requested behavior this value will be initialized to NULL. This event should be unique for each module in order to permit selective quitting of stuck modules. The loader is responsible for closing moduleQuit when appropriate.

(U//FOUO) **pipeName** – The null-terminated name of a bi-directional, byte oriented Windows Named Pipe. The loader is responsible for creating this pipe. The name of the pipe must be random and innocuous. The loader is responsible for shipping any data written to this pipe back to the user. Once at the loader's C2 component the data will be passed through a Unix named pipe to the module C2 handler. Data generated by the module's C2 handler can be conveyed back over the Unix pipe to the loader's C2 component and from there to the module via the pipe with the loader. The module may exchange data with a module-author supplied process on the user's computer over this pipe. The format of data over this pipe is module-specific and defined by the module. When not required by the requested behavior this value will be initialized to NULL. More information on the usage context of this pipe is available in sections 5.3.1 and 5.4.1 below.

(U//FOUO) Pipes are used rather than callback functions to allow for modules to continue running after their loader has quit.

4. (U//FOUO) Metadata

(U//FOUO) Shipped along with the module's DLL file is a metadata file containing several important descriptors to allow the loader's C2 component to perform some initial lightweight validation and error checking. The metadata file must have the same name as the module with the case sensitive ".META.xml" suffix (e.g., for foo.dll the metadata file would be foo.dll.META.xml). An example metadata file follows:

```
<?xml version=”1.0” encoding=”UTF-8” ?>
<module specversion=3>
  <filename>foomodule.1.2.3.dll</filename>
  <hash
type="sha256">8877898D943AF341DA12675A8416BA69A8951713E0466C6D185A51D01
0FC70EE</hash>
  <operatingsystems>
    <OS>w5.1 SP3</OS>
    <OS>w6.1 SP0</OS>
    <OS>w6.1 SP1</OS>
  </operatingsystems>
  <architecture>x86_64</architecture>
  <ordinal>777</ordinal>
  <featuresets>
    <set>Fire</set>
    <set>Collect</set>
    <set>Interact</set>
   </featuresets>
  <quitbehavior willquit="False">WARNING!!!

Module cannot be unloaded without crashing the target. Module has now
stopped execution and will be unloaded on shutdown.</quitbehavior>
  <exports>
    <function>
      <name>screenshot</name>
      <cmdlineprefix>do_foo –x</cmdlineprefix>
      <feature>Collect</feature>
      <help>Takes a screenshot of the active window</help>
    </function>
    <function>
      <name>sshotall</name>
      <cmdlinePrefix>qux –-bar</cmdlineprefix>
      <feature>Interact</feature>
      <help>Takes screenshots of all desktops</help>
    </function>
  </exports>
</module>
```

(U//FOUO) Loader authors must ensure that their C2 component ingests this file and verifies that the module and loader both support the request feature set as well as much of the target details as possible (e.g., that the architecture of the module and target match). The loader is not responsible for validating the user supplied arguments against a `function` entry.

(U//FOUO) To support multiple operating systems it is possible that a given module may require multiple binary builds. Because metadata files are tied in a particular build and not a particular tool it is expected that each binary build will have its own metadata file.

(U//FOUO) To allow developers to add additional tracking information to metadata files as they are processed loaders must ignore unknown tags under the `<reserved>` tag. Any information or tags contained within the `<reserved>` tag is not part of this specification. Any developer-specific information should be prefixed to avoid collisions and must be unclassified in nature.

4.1. (U//FOUO) Metadata Elements and Attributes

(U//FOUO) **`specversion`** – This **`module`** attribute specifies the version of the ICE specification implemented by the module.

(U//FOUO) **`filename`** – The name of the file this metadata file describes.

(U//FOUO) **`hash`** – The cryptographic hash of the file this metadata file describes. Loaders do not need to verify this information. Valid type settings are "`md5`" and "`sha256`". If a loader wishes to verify this value verification should be performed by the loader's C2 component and not the on-target component.

(U//FOUO) **`operatingsystems`** – A list of **OS** entries describing the operating systems supported by this module. A regular expression describing valid values is "w[0-9]\.[0-9] SP[0-9]". The first two numbers are the major and minor version of windows supported; the final number is the service pack applied, 0 if none are applied.

(U//FOUO) **`architecture`** – The ISA the module has been compiled for. Valid values are "`x86`" and "`x86_64`" ("`x86_64`" refers to the architecture commonly known as amd64, x64, or 64 bit). If a module supports more than one architecture it must supply a separate metadata file for each. The use of modules and loaders on non-native architectures is not supported. Particularly, there is no support for injecting a module into a 32 bit process on a native 64 bit OS. This is to limit the development burden of a very rare use case.

(U//FOUO) **`ordinal`** – The ordinal number of the module's ICE function. This is provided here to allow ordinal number diversity and to permit disguising the real ICE entry point among several innocuous exports.

(U//FOUO) **`featuresets`** – A list of **set** entries describing the feature sets support by this module. Valid values are "`Fire`", "`Forget`", "`Collect`", and "`Interact`".

(U//FOUO) **`quitbehavior`** – A very small number of Fire and Collect or Fire and Interact modules will be unable to quit when directed by the loader (e.g., if due to the functionality of the module unloading would cause a BSOD). If a module absolutely cannot quit when directed it must set **`willQuit`** to `False`. The provided string value will be presented to the user via the loader's C2 component to explaining that the module will not unload and why.

(U//FOUO) Modules must make every effort to quit when directed by the loader and this behavior must be used only when absolutely necessary. When a module must refuse the loader's quit direction it should make every effort to stop all functionality and render itself as inert as possible (without interfering with the natural functionality of the system). If **`willQuit`** is False the loader should not wait on `*moduleThread` during exit. The module should still, however, update `*moduleThread` as defined to allow for the possibility that the module gets unloaded by the operating system.

(U//FOUO) **`exports`** – (Optional) A list of `function` entries to provide assistance to the user via the loader's C2 component. This entry (and every `function` entry) is purely optional for both the loader and the module. If supplied, and supported by the loader, this allows a module's functionality to be presented to the user as a native command of the loader. Each `function` entry provides a **`name`** (the command the user will type), a **`cmdlineprefix`**, **`feature`** and a **`help`** string (displayed to the user when the loader's equivalent of `help functionname` is typed). `cmdlineprefix` is appended with the user's provided arguments to construct a true `cmdline` for execution by the module (e.g., in the above example metadata file an user could type "`screenshot -p 1185`" which would be expanded by the loader's C2 to a `MODULE_REMOTE_ARGS cmdline` of "`0 do_foo -x -p 1185`". The **`feature`** tag indicates the minimum feature set required for this function to operate.

5. (U//FOUO) Feature Sets

5.1. (U//FOUO) Fire

(U//FOUO) The Fire feature set permits memory injection of code which is executed in a single function call. Execution of the module may not continue past the return of the `MemoryLoad` function and no interaction is possible other than the return code from `MemoryLoad`. If the loader attempts to quit prior to the module's return from `MemoryLoad` the loader should not terminate until the module returns.

(U//FOUO) The Fire feature set requires and may utilize the following fields in `MODULE_REMOTE_ARGS`: `version`, `hModule`, `cmdline`, `behavior`.

5.2. (U//FOUO) Fire and Forget

(U//FOUO) The Fire and Forget feature set permits memory injection of code which is executed, possibly continuing beyond the return of the `MemoryLoad` function. No interaction is possible other than the return code from `MemoryLoad` and the communication of execution status via `*moduleThread`. Note that `moduleQuit` is not available for Fire and Forget modules. If the loader attempts to quit prior to the module's return from `MemoryLoad` the loader should not terminate until the module returns; if the module signals continued execution via `*moduleThread` the module will not be interrupted and the loader should quit and leak the module's memory and associated handles.

(U//FOUO) The Fire and Forget feature set requires and may utilize the following fields in `MODULE_REMOTE_ARGS`: `version`, `hModule`, `cmdline`, `behavior`, `moduleThread`.

5.3. (U//FOUO) Fire and Collect

(U//FOUO) The Fire and Collect feature set permits memory injection of code which is executed, possibly continuing beyond the return of the `MemoryLoad` function, and has an output path to the user. The Fire and Collect set also permits the loader to trigger the module to quit via `moduleQuit`.

(U//FOUO) The Fire and Collect feature set requires and may utilize the following fields in MODULE_REMOTE_ARGS: version, hModule, cmdline, behavior, moduleThread, moduleQuit, pipeName.

5.3.1. (U//FOUO) Output Pipe

(U//FOUO) pipeName is the null-terminated name of a Windows Named Pipe, suitable for use with WriteFile and related functions. The format of data exchanged over this pipe is controlled by the module author. The module may write arbitrary data to the pipe in the course of its execution, it is the loader's responsibility to convey the written data to the user for analysis. Once back on the C2 computer, the contents of pipeName will be sent over a Unix pipe attached to the module's C2 handler process. This allows, for example, a screenshot module to write screenshots directly to the user's disk while bypassing the remote disk. When executing in Fire and Collect mode the module should consider this a write-only pipe. The module will not receive any data input on this pipe.

(U//FOUO) The loader is responsible for closing the pipe if it must exit prior to the module. The module must detect this closure and cease writing to this pipe in that situation. The loader is also responsible for closing this pipe after module execution ends.

(U//FOUO) The loader may cache output for transmission (e.g., to chunk output). If data must be cached to disk it must be encrypted prior to being written to disk. The loader's C2 component will not interact in any way with the data sent over this pipe except to pass it to the module's C2 component.

5.4. (U//FOUO) Fire and Interact

(U//FOUO) The Fire and Interact feature set permits memory injection of code which may continue execution beyond the return of the MemoryLoad function while providing output to and receiving input from the user. The Fire and Interact set also permits the loader to trigger the module to quit via moduleQuit.

(U//FOUO) The Fire and Interact feature set requires and utilizes the following fields in MODULE_REMOTE_ARGS: version, hModule, cmdline, behavior, moduleThread, moduleQuit, pipeName.

5.4.1. (U//FOUO) Input and Output Pipe

(U//FOUO) pipeName is the null-terminated name of a Windows Named Pipe, suitable for use with WriteFile, ReadFile and related functions. The format of data exchanged over this pipe is controlled by the module author. In addition to the output as in section 5.3.1 the Fire and Interact mode pipe supports input. Input is generated by the module's C2 component (e.g., based on user interaction with the module C2 component) which will send data over a Unix pipe to the loader's C2 component on the C2 computer. The loader's C2 component then conveys the information to the loader which will then convey data to the pipe on the remote computer for the module's ingestion. This allows, for example, a module to tunnel its communications over the loader rather than opening a new socket.

(U//FOUO) The loader is responsible for closing this pipe if it must exit prior to the module. The module must detect this closure and cease reading from the pipe in that situation. The loader is also responsible for closing the pipe after module execution ends.

(U//FOUO) The loader's C2 component will not interact in any way with the data sent over this pipe except to pass it between the module's C2 component and the module.

6. (U//FOUO) Communicating Status

(U//FOUO) If a Fire and Forget (or higher) module wishes to remain loaded after returning from its ordinal function then it must assign a thread handle to *`moduleThread`. After the module returns from its ordinal function, the loader will wait on this thread, if assigned, and clean up after the thread exits. This functionality exists for the purposes of avoiding a memory leak in long running loaders.

(U//FOUO) Modules create their own long-lived threads rather than rely on any threads created by the loader to avoid a possible access violation. For example, an alternate specification where the module's main thread was the same as the loader's `MemoryLoad` calling thread would return to a now-freed address if the loader quit during the module's execution. This scenario could result in substantial alerting behavior.

(U//FOUO) This behavior is designed to allow the loader to free the memory that the module is initially executed from. If a module's behavior involves process migration or other execution migration methods that permit the initial execution memory to be deallocated then the module does not need to communicate status in this way.

6.1. (U//FOUO) Loader Responsibilities

(U//FOUO) The loader shall initialize the memory pointed to by `moduleThread` (that is, *`moduleThread`) to NULL. After the module returns from `MemoryLoad` the loader may check to see if *`moduleThread` has been set to a different value and, if so, may have a (non-main) thread block on that thread. After the loader observes that *`moduleThread` has exited the loader will unload the module, zero, and then free the memory in which the module was loaded. The loader is responsible for closing `*moduleThread` (and `moduleQuit` if applicable) once these steps have been completed.

(U//FOUO) The loader is responsible for returning the exit code from `*moduleThread` to the user. This might be done well after the user initially executes the module and is provided to allow for an additional communication and debugging mechanism should a long-running module quit unexpectedly.

(U//FOUO) The loader is only responsible for freeing the memory in which the module's executable code resides; it is not responsible for freeing any module allocated memory.

6.2. (U//FOUO) Module Responsibilities

(U//FOUO) If the module desires to remain loaded after the module's `MemoryLoad` function returns then the module should take the following actions. At the beginning of execution the module shall check that *`moduleThread` has been set to `NULL`, if it has not then the module should not perform any further status communication. After a successful check of *`moduleThread` the module should set *`moduleThread` to a thread handle. This should be done prior to the ordinal function returning. Because the loader will unload the module when the assigned thread exits it is suggested that modules assign a main thread handle to `*moduleThread`. The exit code from `*moduleThread` will be conveyed to the user and has no intrinsic meaning to this specification.

(U//FOUO) The module remains responsible for freeing all memory it allocates.

7. (U//FOUO) Return Values

(U//FOUO) After a module's MemoryLoad function returns, and unless otherwise indicated via *moduleThread, the loader assumes the module is done executing and will unload the module and its dependencies and zero the space the module occupied. The following return values are defined to permit this behavior to be overridden in erroneous circumstances. Return values are interpreted by first casting the DWORD value as a signed integer.

(U//FOUO) A positive return value indicates no critical error has occurred in the context of this specification. Note that this may still mean the module has failed to complete its function for some external reason. The meaning of the positive return codes is defined by the module and has no intrinsic meaning in this specification. Negative return codes greater than or equal to -128 (e.g -122) indicate a non-critical error has occurred and that the module can be unloaded. Error codes in the non-critical range not specified in this document are reserved for future use and should not be used. A negative return value less than -128 (e.g. -129) indicates a critical error. The loader will not attempt to unload or otherwise interact with a module which has experienced a critical error.

```
#define ICE_ERROR_SUCCESS           0
#define ICE_ERROR_WRONG_BEHAVIOR   -1
#define ICE_ERROR_WRONG_ARGUMENTS -2
#define ICE_ERROR_WRONG_VERSION    -3
```

(U//FOUO) If a module is called with requested behavior other than that actually supported by the module it must return ICE_ERROR_WRONG_BEHAVIOR. If a module is called with incorrect or invalid MODULE_REMOTE_ARGS structure elements it must return ICE_ERROR_WRONG_ARGUMENTS. If a module is called with a version which it does not support then it must return ICE_ERROR_WRONG_VERSION. If a module has accomplished its function without error it must return ICE_ERROR_SUCCESS. All other return values are controlled by the module author.

(U//FOUO) If a module's MemoryLoad returns ICE_ERROR_SUCCESS or a positive value then the module can be unloaded and freed as per the defined behavior. If a module returns a non-critical error code (represented by the range [-1, -128]) including ICE_ERROR_WRONG_BEHAVIOR, ICE_ERROR_WRONG_ARGUMENTS, or ICE_ERROR_WRONG_VERSION then the module can be immediately unloaded and freed regardless of the requested behavior (ignoring the value of *moduleThread). If MemoryLoad returns any other value the loader should assume a critical error and leave the module alone, leaking its memory.

(U//FOUO) If the loader is unable to unload, zero, and free the module's memory (e.g., due to a module thread that has continued to run) then the loader's C2 will alert the user.

(U//FOUO) [DEPRECATION WARNING] The following return logic is legacy behavior included here for interoperability testing. Modules compliant with version 3 or higher of this specification must not implement this behavior (i.e., returning ERROR_UNABLE_TO_UNLOAD_MEDIA will *not* tell the loader you wish to remain loaded). Loaders compliant with version 3 or higher of this specification may include this feature if memory leaks are not a concern in the loader's operating environment.

(U//FOUO) If a module compliant with version 2 of this specification desires to remain loaded (e.g. a module that installs hooks in the user-mode process but does not execute code itself) the `MemoryLoad` function should return the special error code `HRESULT_FROM_WIN32 (ERROR_UNABLE_TO_UNLOAD_MEDIA)`. In this case the module and any dependencies it triggered can no longer be unloaded and will remain in memory until the process exits.

8. (U//FOUO) Load and Execute Loop

(U//FOUO) Loaders shall work with modules on a Load, Execute, Unload basis. Loaders shall not attempt to execute a module more than once before unloading. This is defined to avoid ambiguity about module reentrancy and to support loader optimizations.

9. (U//FOUO) Module Caching

(U//FOUO) Loaders may cache modules on the remote computer separately in non-loaded, preferably encrypted form. Modules may only be cached to disk in an encrypted form. In all cases the loader's C2 component must explicitly notify the user that caching has been performed and provide a way to purge a module from the cache. This is explicitly permitted to accommodate certain use cases.

10. (U//FOUO) Structured Exception Handling

(U//FOUO) Loaders will not provide fix ups to allow ICE modules to use Structured Exception Handling (SEH). If an ICE module wishes to use SEH it must perform the fix ups itself.

11. (U//FOUO) Vectorized Exception Handling

(U//FOUO) Loaders will not provide fix ups to allow ICE modules to use Vectorized Exception Handling (VEH). If an ICE module wishes to use VEH it must perform the fix ups itself.

12. (U//FOUO) Pipe Opening and Closing Order

(U//FOUO) The pipe used for Fire and Collect and Fire and Interact shall be created and destroyed in the following order:

12.1. (U//FOUO) Loader creates the pipe.

12.2. (U//FOUO) Loader opens loader-side handle to the pipe

12.3. (U//FOUO) Loader passes pipe name to module via MemoryLoad invocation

12.4. (U//FOUO) Module opens module-side handle to pipe

12.5. (U//FOUO) Module returns from MemoryLoad

12.6. (U//FOUO) Module and loader communicate via pipe

12.7. (U//FOUO) Module prepares to end execution

12.8. (U//FOUO) Module closes module-side handle to pipe

12.9. (U//FOUO) Module exits

12.10. (U//FOUO) Loader detects that module has closed module-side handle to pipe and closes loader-side handle in response.

12.11. (U//FOUO) Once all handles to the pipe are closed, loader frees the pipe.

(U//FOUO) In addition to the above order both the loader and the module must be able to handle a pipe being closed unexpectedly. In this situation the loader or module must close its own open handle, and react as appropriate (i.e., a module with other functionality should continue running, a module with no other functionality should quit).

13. (U//FOUO) Dependency Resolution and Ordinals

(U//FOUO) Modules which export dependency information via ordinals rather than function names must ensure that their metadata file correctly correlates OS version with these ordinal dependencies. Modules are encouraged to ship multiple metadata files if necessary. For example, in a situation where a module claims support for Windows XP SP3 and Windows 7 SP1 and depends, via ordinal, on a particular function that has changed ordinal values it is a bug in the module, not the loader.

14. (U//FOUO) Hooking From Multiple Directions

(U//FOUO) In the course of the sponsor's business it is possible that multiple tools may attempt to hook the same OS or third party function at the same time resulting in a race condition during unhooking. Because authors cannot exhaustively test all combinations the sponsor assumes responsibility for eliminating this variable. To enable the sponsor to perform this analysis, authors are required to document any system or third-party functions they hook and the purpose of the hook.

(U//FOUO) This guidance applies especially to the function `SetUnhandledExceptionFilter`. All uses of this function must be documented.

15. (U//FOUO) Behaviors to Avoid

(U//FOUO) Certain functions can have an adverse affect on the process or overall system if called by a module injected into a system process. As a result, modules should not call any functions that alter process-level state (e.g., `SetDllDirectory, SetErrorMode`). If a module must call one of these functions then the function called and the purpose of its call must be documented.

Appendix A. (U//FOUO) Version History

(U//FOUO) Version 1: Legacy

(U//FOUO) Version 2: Fire and Forget Specification.

- Initial publication (as Fire and Forget Specification v1.0).
- `MODULE_REMOTE_ARGS` contains `version`, `hModule`, moduleSize, and `cmdline`.
- `MemoryLoad` returns HRESULT.

(U//FOUO) Version 3: Fire and Interact Specification.

- Defined additional feature sets “Fire”, “Fire and Collect”, and “Fire and Interact”.
- Renamed to In-Memory Code Execution suite.
- Altered document version number to track struct version number.
- Added communicating execution status between module and loader.
- Added metadata file to document module capabilities.
- Added support for arbitrary ordinal number.
- `MODULE_REMOTE_ARGS` contains `version`, `hModule`, `cmdline`, `behavior`, `moduleThread`, `moduleQuitMutex`, `textOut`, `binOut`, `textIn`, and `binIn`.
- `MemoryLoad` returns DWORD.

(U//FOUO) Version 3 revision 2

- Clarified several subtle details in response to developer feedback from partial implementations.
- Altered `moduleQuitMutex` to `moduleQuit` and changed its type from a Mutex to a Windows Event.
- Added sections 11 through 15.
- `MODULE_REMOTE_ARGS` contains `version`, `hModule`, `cmdline`, `behavior`, `moduleThread`, `moduleQuit`, `textOut`, `binOut`, `textIn`, and `binIn`.

(U//FOUO) Version 3 revision 3

- Based on implementer feedback, consolidated the four pipes into a single bi-directional pipe.
- Based on implementer feedback, caused the module handler process to be invoked by the user rather than the loader’s C2 component. This is 2.1.
- Based on implementer feedback, inserted 2.4.2.

- Added the figures in section 1 to clarify the relationships between the different feature sets and the execution context of each component.
- Clarified that ICE is primarily designed for in-memory execution and some remote process injection components have been deferred.
- Clarified that the loader thread which calls `DllMain` and the loader thread which calls `MemoryLoad` may not be the same thread.
- Required Fire and Collect and Fire and Interact pipes to be bidirectional, byte oriented Windows Named Pipes and that the name of the pipe must be random and innocuous.
- Clarified that pipes are used instead of callback functions to avoid a possible access violation.
- Clarified why modules create their own long-lived threads rather than rely on the `MemoryLoad` calling thread.
- Clarified that loaders must expect Fire and Forget or higher modules to remain running after the loader quits. This is 2.4.3.
- Clarified that the loader's C2 component will never interact with the module's pipe data other than to pass it to the module's C2 component.

(U//FOUO) Version 3 final

- Based on implementer feedback, replaced named pipe with Unix domain socket for communication on C2 system between loader and module handler.
- Based on implementer feedback added minimum feature set for each exported function
- Based on design feedback defined ranges for critical and non-critical error return codes for module `MemoryLoad` function.
- This is the final revision of ICE version 3.

Appendix B. (U//FOUO) Metadata XML DTD

(U//FOUO)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!ELEMENT module (filename, hash, operatingsystems, architecture,
ordinal, featuresets, quitbehavior, exports?, reserved?)>
<!ATTLIST module
   specversion (2|3) #REQUIRED
>
<!ELEMENT filename (#PCDATA)> <!-- The filename of the described -->
                        <!-- module adjacent to the metadata file -->
<!ELEMENT hash (#PCDATA)>          <!-- [0-9a-f]+ -->
<!ATTLIST hash
   type (md5|sha256) #REQUIRED
>
<!ELEMENT operatingsystems (OS)+>
<!ELEMENT OS (#PCDATA)>            <!-- w[56]\.[0-9] SP[0-9] -->
<!ELEMENT architecture (#PCDATA)>  <!-- x86(_64)? -->
<!ELEMENT ordinal (#PCDATA)>       <!-- [0-9]+ -->
<!ELEMENT featuresets (set)+>
<!ELEMENT set (#PCDATA)> <!—- (Fire|Forget|Collect|Interact) -->
<!ELEMENT quitbehavior (#PCDATA)>  <!-- String to display to the user
when willquit=False -->
<!ATTLIST willquit
  willquit (True|true|1|False|false|0) #REQUIRED
>
<!ELEMENT exports (function)+>
<!ELEMENT reserved (#PCDATA)>    <!--Additional tags are acceptable -->
<!ELEMENT function (name, cmdlineprefix, feature, help)>
<!ELEMENT feature (#PCDATA)>    <!-- (Fire|Forget|Collect|Interact) -->
<!ELEMENT name (#PCDATA)>          <!-- [a-zA-Z0-9_-]+ -->
<!ELEMENT cmdlineprefix (#PCDATA)> <!-- Inserted prior to remainder -->
               <!-- of user's input as cmdline in module arg struct -->
<!ELEMENT help (#PCDATA)>          <!-- Human readable text -->
                <!-- displayed when user types "help" or equivalent -->
```

Appendix C. (U//FOUO) Metadata XML XSD

(U//FOUO)

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
targetNamespace=”urn:module” xmlns:mod=”urn:module”>

  <xs:element name=”module” type=”mod:ModuleType” />

  <xs:simpleType name=”SpecVerType”>
    <xs:restriction base=”xs:string”>
      <xs:pattern value=”3” />
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name=”ModuleType”>
    <xs:all>
      <xs:element name=”filename” type=”xs:string” minOccurs=”1”
maxOccurs=”1” />
      <xs:element name=”hash” type=”mod:FileHashType” minOccurs=”1”
maxOccurs=”1” />
      <xs:element name=”operatingsystems” minOccurs=”1” maxOccurs=”1”>
        <xs:complexType>
          <xs:sequence>
            <xs:element name=”OS” type=”mod:OsType” minOccurs=”1”
maxOccurs=”unbounded” />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name=”architecture” type=”mod:ArchType” minOccurs=”1”
maxOccurs=”1” />
      <xs:element name=”ordinal” minOccurs=”1” maxOccurs=”1”>
        <xs:simpleType>
          <xs:restriction base=”xs:unsignedShort”>
            <xs:minInclusive value=”1” />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name=”featuresets” minOccurs=”1” maxOccurs=”1”>
        <xs:complexType>
          <xs:sequence>
            <xs:element name=”set” type=”mod:FeatureSetValue”
minOccurs=”1” maxOccurs=”unbounded” />
          </xs:sequence>
        </xs:complexType>
        <xs:unique name=”uniqueFunctionSetVal”>
          <xs:selector xpath=”mod:set” />
          <xs:field xpath=”.” />
        </xs:unique>
      </xs:element>
      <xs:element name=”quitbehavior” type=”mod:QuitBehaviorType”
minOccurs=”1” maxOccurs=”1” />
      <xs:element name=”exports” minOccurs=”0” maxOccurs=”1”>
        <xs:complexType>
          <xs:sequence>
            <xs:element name=”function” type=”mod:ExportedFunctionType”
minOccurs=”1” maxOccurs=”unbounded” />
          </xs:sequence>
        </xs:complexType>
        <xs:unique name=”uniqueExportedFunctionVal”>
          <xs:selector xpath=”mod:function” />
          <xs:field xpath=”./name” />
        </xs:unique>
      </xs:element>
      <xs:element name=”reserved” minOccurs=”0” maxOccurs=”1” />
    </xs:all>
    <xs:attribute name=”specversion” type=”mod:SpecVerType”
use=”required” />
  </xs:complexType>

  <xs:simpleType name=”HashType”>
    <xs:restriction base=”xs:string”>
      <xs:pattern value=”md5|sha256” />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name=”FileHashData”>
    <xs:restriction base=”xs:string”>
      <xs:pattern value=”[0-9a-f]+” />
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name=”FileHashType”>
    <xs:simpleContent>
      <xs:extension base=”mod:FileHashData”>
        <xs:attribute name=”type” type=”mod:HashType” use=”required” />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:simpleType name=”OsType”>
    <xs:restriction base=”xs:string”>
      <xs:pattern value=”w[56]\.[0-9] SP[0-9]” />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name=”ArchType”>
    <xs:restriction base=”xs:string”>
      <xs:pattern value=”x86|x86_64” />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name=”FeatureSetValue”>
    <xs:restriction base=”xs:string”>
```

```
      <xs:pattern value=”Fire|Forget|Collect|Interact” />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name=”WillQuitType”>
    <xs:restriction base=”xs:string”>
      <xs:pattern value=”True|true|1|False|false|0” />
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name=”QuitBehaviorType” mixed=”true”>
    <xs:attribute name=”willquit” type=”mod:WillQuitType”
use=”required” />
  </xs:complexType>

  <xs:complexType name=”ExportedFunctionType”>
    <xs:all>
      <xs:element name=”name” type=”xs:string” />
      <xs:element name=”cmdlineprefix” type=”xs:string” />
      <xs:element name=”feature” type=”mod:FeatureSetValue” />
      <xs:element name=”help” type=”xs:string” />
    </xs:all>
  </xs:complexType>

</xs:schema>
```

Appendix D. (U//FOUO) Future Work

(U//FOUO) The current version of this specification does not cover all discussed ideas. This section records some of those whose implementation is deferred to some future version in order to minimize implementation costs of this standard:

- (U//FOUO) A metadata field containing a module’s list of required privilege levels.
- (U//FOUO) A metadata field containing a list of processes that the module must execute within.