**DotNetVault**

Synchronization Library and Static Analyzer for C# 8.0

Project Description

Author: Christopher P. Susie

# Table of Contents


1. Introduction. ..... 6
   a. Currently used synchronization methods ..... 6
   b. Problems with current lock-based mechanisms ..... 7
      i. Primary problem with current mechanisms is they protect data only when programmers follow convention; Also, try … finally syntax is error prone ..... 7
      ii. Atomic operations are highly useful alternative but not easy to understand and scope of usefulness limited compared to locks ..... 8
      iii. C#'s lock mechanism is not timed when used in its RAII form and is bug prone when used in its try…finally form ..... 8
      iv. C#'s Monitor Lock mechanism is recursive ..... 8
      v. Carefully crafted objects can be the most effective solution, but these are often not possible or maintainable by all given changing requirements ..... 12
   c. DotNetVault isolates protected data and prevents access to it without first obtaining a lock ..... 12
      i. C# 8's Disposable ref struct is used to isolate obtained locks on the stack and ensure prompt release in all cases ..... 13
      ii. Static analysis prevents leakage or mingling of shared mutable state ..... 14
2. Prerequisites ..... 15
3. Installation ..... 15
4. Usage Guide ..... 15
   a. Concept of vault-safety ..... 15
   b. Overview of Tools ..... 17
      i. *Vaults* ..... 17
      ii. *LockedResources* ..... 17
   c. Vaults In-Depth ..... 19
      i. Underlying Synchronization Mechanisms ..... 19
      ii. Atomic Vaults ..... 19
      iii. Monitor Vaults ..... 19
      iv. ReadWrite Vaults ..... 20

v. Functionality Common to All Vaults (intended for public consumption) ..... 22
vi. *Basic Vaults* ..... 27
vii. Mutable Resource Vaults ..... 28
viii. *CustomizableMutableResourceVault<T>* ..... 31
d. LockedResources In-Depth ..... 32
*i.* Common Functionality ..... 32
*ii.* *Vaults and their LockedResources* ..... 32
*iii.* Suggestion Regarding Declaration of Locked Resource Objects (i.e., Use "*var*") ..... 34
*iv.* *Locked Resource Objects of Basic Vaults* ..... 35
( *LockedVaultObject<BasicVault<T>, T>* and *LockedMonVaultObject<BasicMonitorVault<T>, T> )* ..... 35
*v.* Locked Resource Objects of Mutable Resource Vaults ..... 37
5. Static Analyzer Rules ..... 46
a. DotNetVault_UsingMandatory ..... 46
b. DotNetVault_VaultSafe ..... 46
c. DotNetVault_VsDelegateCapture ..... 47
d. DotNetVault_VsTypeParams ..... 47
e. DotNetVault_NotVsProtectable ..... 47
f. DotNetVault_NotDirectlyInvocable ..... 50
g. DotNetVault_UnjustifiedEarlyDispose ..... 50
*i.* *EarlyReleaseReason.DisposingOnError* ..... 50
ii. *EarlyReleaseReason.CustomWrapperDispose* ..... 51
h. DotNetVault_NoExplicitByRefAlias ..... 52
6. Attributes ..... 52
a. VaultSafeAttribute ..... 52
b. UsingMandatoryAttribute ..... 53
c. VaultSafeTypeParamAttribute ..... 53
d. NoNonVsCaptureAttribute ..... 54
e. NotVsProtectableAttribute ..... 54
f. NoDirectInvokeAttribute ..... 54

g. EarlyReleaseAttribute ..... 55
h. EarlyReleaseJustificationAttribute ..... 55
i. BasicVaultProtectedResourceAttribute ..... 55
7. Known Flaws and Limitations ..... 56
a. Table of Known Issues ..... 56
b. Example Code Showing Problems ..... 57
8. Licensing ..... 62
a. Software License ..... 62
b. Documentation License ..... 62
c. Author Contact Information ..... 63

# Table of Figures

Figure 1 – RAII style Lock and Try…Finally Equivalent ..... 7
Figure 2 -- Typical Wrapper Around Well-Known Interface ..... 9
Figure 3 – Typical Flawed Attempt at Post Hoc Thread Safety ..... 10
Figure 4 – Demonstrates Lack of State Consistency Between Calls. ..... 11
Figure 5 – Typical Mechanism for Achieving Thread Safety After the Fact ..... 11
Figure 6 – Public Properties Common to All Vaults ..... 22
Figure 7 – Public Methods Common to All Vaults ..... 23
Figure 8 – Lock and Spinlock Overloads ..... 27
Figure 9 – Correct and Incorrect Creation of MutableResourceVault ..... 29
Figure 10 – More Elaborate Correct and Incorrect Creation of MutableResourceVault ..... 30
Figure 11 -- Return by Reference and Ref Local Alias Prohibition ..... 36
Figure 12 -- Output from Figure 11 ..... 37
Figure 13 -- Special Delegates Used By LockedVaultMutableResource Objects to Prevent Leakage and Mingling of State ..... 38
Figure 14 -- (cont'd) Special Delegates Used By LockedVaultMutableResource Objects to Prevent Leakage and Mingling of State ..... 39
Figure 15 -- VaultQuery Demonstration ..... 41
Figure 16 -- VaultQuery Demo Output ..... 41
Figure 17 -- VaultAction Demonstration ..... 42
Figure 18 -- VaultAction Demo Output ..... 42
Figure 19 – VaultMixedOperation Demonstration ..... 43
Figure 20 -- VaultMixedOperation Demo Output ..... 44
Figure 21 – Demonstration of Extension Methods to Simplify Usage ..... 45

Figure 22 -- Output of Extension Method Demo ..... 45
Figure 23 -- vault-safe Convenience Wrappers ..... 48
Figure 24 – Usage of Vs Convenience Wrappers ..... 49
Figure 25 -- Usage Wrapper Demo Output ..... 49
Figure 26 – If the resource is not manually released before exceptions rethrown, it will be forever inaccessible. ..... 51
Figure 27 – Shows how to annotate the Dispose method of custom locked resource objects. ..... 51
Figure 28 -- -- Contents of Whitelist.txt ..... 52
Figure 29-- Contents of condit_generic_whitelist.txt ..... 53
Figure 30 – Double Dispose (Known Flaw #1 -- FIXED) ..... 57
Figure 31 – Bad Extension Method (Known Flaw #2 -- FIXED) ..... 58
Figure 32 – Bad Type Inherently Leaks (Known Flaw #3) ..... 59
Figure 33 – Shows Bug 64 Fix ..... 60
Figure 34 -- Demonstrates Bug 76 and its Fix ..... 61

# 1. Introduction.

DotNetVault is a library and static code analysis tool that makes managing shared mutable state in multi-threaded applications more manageable and less error prone. Where errors do still occur, they are easier to locate and identify.

Managing shared mutable state in multithreaded environments is one of the most difficult feats to accomplish in complex application development. The tools available generally rely on each programmer's knowledge of the synchronization mechanisms employed. Each programmer in the project must know and follow them always, even when that programmer's task or expertise lies outside threading issues. Even highly experienced multi-threading programmers make mistakes because focus on the domain area or a difficult unrelated task may divert his or her attention. These mistakes are difficult to reproduce, diagnose, find and correct. They rank among the costliest bugs in software projects.

## *a. Currently used synchronization methods*

One way to eliminate issues of synchronization is to eliminate the use of shared mutable state. If all state is mutable but not shared or shared but not mutable, worries about thread synchronization disappear. For most application development, mutability of shared state can be *reduced* but rarely *eliminated*. Furthermore, the ubiquity of reference types in C# works to thwart attempts at preventing sharing. Thus, in C#, programmers are frequently left with traditional synchronization primitives that require a portion of the engineer's mental focus, leaving her unable to focus entirely on solving the problem at hand. The most frequently used primitive in C# to synchronize state is the *lock* statement.[1] Lock is a simple RAII[2]-like mechanism around a more convoluted and bug-prone syntax involving *try … finally*:

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/lock-statement

[2] RAII stands for Resource Acquisition is Initiation. It can also described as Scope-Based Resource Management. It is the primary method used in modern C++ to manage resource lifetime and is also the method employed by the Rust programming language. Its principle is simple: when an object is initialized it acquires all its resources and establishes all invariants – making it impossible for an uninitialized or invalid object to exist. Equally, and perhaps more importantly, when an object's lifetime ends (typically bound to scope), at that moment it releases any resources it acquired. https://en.cppreference.com/w/cpp/language/raii; http://www.stroustrup.com/bs_faq2.html#finally

The lock statement is of the form

```
lock (x)
{
    // Your code...
}
```

where x is an expression of a reference type. It's precisely equivalent to

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

*Figure 1*[3] – RAII style Lock and Try…Finally Equivalent

There are also other similar synchronization mechanisms available to the C# programmer: The *System.Mutex*[4] class, *ReaderWriterLockSlim*[5], *SpinLock*[6] and many others. In addition to these locking mechanisms, there are lock-free thread safety mechanisms that provide thread-safety through atomic operations.[7]

### *b. Problems with current lock-based mechanisms*

i. Primary problem with current mechanisms is they protect data only when programmers follow convention; Also, try … finally syntax is error prone

Of the lock-based synchronization mechanisms provided, there are many drawbacks. The primary drawback is that they rely on programmer discipline to only access the protected objects when the lock is obtained. A related drawback is that, except for *lock* which incorporates guaranteed scope-based release of the lock, the programmer must remember to release the lock even in exceptional cases, but not release it too soon. Most, if not all, of the synchronization mechanisms other than *lock* do not provide an

[3] v. *supra*, #1.
[4] https://docs.microsoft.com/en-us/dotnet/api/system.threading.mutex?view=netframework-4.8
[5] https://docs.microsoft.com/en-us/dotnet/api/system.threading.spinwait?view=netframework-4.8
[6] https://docs.microsoft.com/en-us/dotnet/api/system.threading.spinlock?view=netframework-4.8
[7] https://docs.microsoft.com/en-us/dotnet/api/system.threading.interlocked?view=netframework-4.8

RAII-like way to do this, so try finally must be employed, which is a convoluted syntax, susceptible to being forgotten or mistakenly implemented.[8]

ii. Atomic operations are highly useful alternative but not easy to understand and scope of usefulness limited compared to locks

Furthermore, atomic operations are no panacea: they are frequently difficult to understand for programmers less experienced with multithreading and offer a much smaller window of atomicity than that provided by the mutex/lock-based synchronization mechanisms. Nevertheless, they do provide a valuable alternative to mutexes when used judiciously. DotNetVault makes extensive internal use of them.

iii. C#'s lock mechanism is not timed when used in its RAII form and is bug prone when used in its try…finally form

The heavy onus on the programmer to refrain from using protected resources (and knowing which resources are protected and which protected resources are associated with which mutex) is far from the only other drawback of the commonly employed mutexes. When multiple mutexes are held simultaneously, the user must be especially disciplined to make sure that *every time* they are acquired, the same relative ordering is applied. Failure to do this can sometimes result in silent deadlock of the program but may not be reliably reproducible. One way to get around this possibility is to set a timeout when you attempt to acquire this mutex: instead of silent deadlock an exception will be thrown, making it easy to identify the problem and correct it. Unfortunately, C# does not provide an out-of-the-box RAII-based mechanism to add a time limit, forcing you to rely on convoluted try-finally constructs as shown above.[9]

iv. C#'s Monitor Lock mechanism is recursive

Another problematic consideration is C#'s usage of recursive mutexes (and similar locking mechanisms). Most mutex mechanisms in C# are recursive, recursive by default or at least optionally recursive. The most used mechanism is the *Monitor.TryEnter* method or its more reliable RAII *lock* shorthand.[10] This type of mutex is, by default, recursive. The primary desirable use-case for a recursive mutex is the adaptation of code

[8] v. #1, *supra.*
[9] v. Figure 1, *supra.*
[10] v. #1, *supra.*

that was not designed to be thread-safe to some semblance of thread-safety, *without changing the API*.[11] This usage, however, should be considered a last resort.[12]

- *Recursive mutexes are intended to be used to adapt non-thread safe objects to a thread-safe context*

To see how the recursive mutex can work to adapt non-thread-safe code to thread-safety while remaining problematic, consider the following custom list wrapper that presumably would add some domain-specific functionality to a standard List<T>:

```
sealed class SomeCustomStringList : IList<string>
{
    public int Count => _wrappedList.Count;
    public bool IsReadOnly => false;

    public string this[int index]
    {
        get => _wrappedList[index];
        set => _wrappedList[index] = value;
    }

    public SomeCustomStringList() => _wrappedList = new List<string>();
    public SomeCustomStringList([NotNull] IEnumerable<string> source) => _wrappedList =
        new List<string>(source ?? throw new ArgumentNullException(nameof(source)));

    public IEnumerator<string> GetEnumerator() => _wrappedList.GetEnumerator();
    public void Add(string item) => _wrappedList.Add(item);
    public void Clear() => _wrappedList.Clear();
    public bool Contains(string item) => _wrappedList.Contains(item);
     public void CopyTo(string[] array, int arrayIndex) => _wrappedList.CopyTo(array,
            arrayIndex);
    public bool Remove(string item) => _wrappedList.Remove(item);
    public int IndexOf(string item) => _wrappedList.IndexOf(item);
    public void Insert(int index, string item) => _wrappedList.Insert(index, item);
    public void RemoveAt(int index) => _wrappedList.RemoveAt(index);
     IEnumerator IEnumerable.GetEnumerator()
            => ((IEnumerable)_wrappedList).GetEnumerator();

    private readonly List<string> _wrappedList;
}
```

*Figure 2* -- Typical Wrapper Around Well-Known Interface

[11] https://en.wikipedia.org/wiki/Reentrant_mutex#cite_note-2
[12] *Id.*

As frequently happens, later in the development of the software, it is discovered or decided that a thread-safe version of this type is required. The following code results is a typical result:

```
public sealed class CustomStringList : IList<string>
{
    public object SyncObject { get; } = new object();

    public int Count { get { lock (SyncObject) return _wrappedList.Count;} }

    public bool IsReadOnly => false;

    public string this[int index]
    {
        get { lock (SyncObject) return _wrappedList[index]; }
        set { lock (SyncObject) _wrappedList[index] = value; }
    }

    public IEnumerator<string> GetEnumerator()
    {
        //can't return an enumerator to this list, have to return an enumerator of a snapshot
        lock (SyncObject)
        {
            var snap = new SomeCustomStringList(_wrappedList);
            return snap.GetEnumerator();
        }
    }


    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

    //rest proceeds by just delegating and wrapping in lock…….
}
```

*Figure 3* – Typical Flawed Attempt at *Post Hoc* Thread Safety

- *While it may achieve this on a syntactic level, it usually fails to do so on a semantic level leading to unmaintainable code*

Note that the *GetEnumerator*() method returns an enumerator to a snapshot copy of the collection rather than an enumerator to the actual collection.[13] Also, the synchronization object is exposed (thus adding to the API but not removing from it or altering its signatures). The reason for the exposure of the synchronization object is shown in this example:

[13] Would all developers on your team think of this? Of those that would, will they realize they need to do it even when under other pressures? Is this type of thing really a good idea anyway?

```
public static void DoStuffWithList([NotNull] CustomStringList list)
{
    if (list == null) throw new ArgumentNullException(nameof(list));

    var idxToFirstJaneInList = list.IndexOf("Jane");
    if (idxToFirstJaneInList > -1)
    {
        list[idxToFirstJaneInList] = "Janey";
    }
}
```

*Figure 4* – Demonstrates Lack of State Consistency Between Calls.

The problem here should be immediately apparent: the lock is released after execution of *IndexOf* is complete then is ***reacquired*** for the call to the set accessor of the indexer property in *list[idxToFirstJaneInList]*. The list's contents may change between the two calls. So, given the recursive nature of *lock*s in C# and the handy public *SyncObject* property exposed by the thread-safe wrapper, the solution presents itself:

```
public static void DoStuffWithList([NotNull] CustomStringList list)
{
    if (list == null) throw new ArgumentNullException(nameof(list));

    lock (list.SyncObject)
    {
        var idxToFirstJaneInList = list.IndexOf("Jane");
        if (idxToFirstJaneInList > -1)
        {
            list[idxToFirstJaneInList] = "Janey";
        }
    }
}
```

*Figure 5* – Typical Mechanism for Achieving Thread Safety After the Fact

Notice that in attempting to preserve the same API for the object, we have really introduced a vexing problem that will propagate to all client code. It is rare that you only do one "operation" on a list: typically, you do multiple operations on a list and each successive operation relies on the previous operations' results still being accurate. Thus, nearly everywhere your new thread-safe implementation is used, you will have to make sure that you obtain a recursive lock. Also, it will not be a compiler error or obviously wrong on inspection of any code block, *as a unit*, if this lock is not first obtained. Recursively obtaining mutexes is not free from a performance penalty compared to non-recursive mutexes.

This approach fails: while it *appears* to preserve the same API the thread-unsafe List had, it does so *only with syntax*. To make the thread-safe version equivalent to the

original version *semantically*, additional actions must be taken by the caller, violating the Liskov substitution principle[14] and leading to bugs and unmaintainable code.

v. Carefully crafted objects can be the most effective solution, but these are often not possible or maintainable by all given changing requirements

One approach superior to the recursive mutex approach is to design a special purpose object to access the list. While this may be superior *in theory*, it requires you to anticipate which sequence of access to the list will be required and to expose them all in the object's public interface. This is more difficult than it sounds in the light of changing requirements and the differing skill levels of project contributors who may be forced to alter the thread safe object as new needs are discovered.

*c. DotNetVault isolates protected data and prevents access to it without first obtaining a lock*

DotNetVault takes its inspiration from the synchronization mechanisms provided by Rust language and the Facebook Folly C++ synchronization library. These synchronization mechanisms observe that the mutex should *own* the data they protect. You literally cannot access the protected data without first obtaining the lock. RAII destroys the lock when it goes out of scope – even if an exception is thrown or early return taken.[15]

In a sense, languages like C++ and Rust that heavily use types with value semantics, incorporate the concept of *ownership* of a resource and provide *move* semantics are more easily able to provide isolation of resources than a language like C# where so many we access so many objects through garbage collected references: these references, thanks to garbage collection, can be shared promiscuously and by their very nature almost beg to be shared. While garbage collection saves us from the nightmare of keeping track of who needs to free all these freely shared references to heap objects[16], it works at cross-purposes to isolation aimed at thread-safety.

---

[14] The Liskov substitution principle requires a derived class to be a drop-in substitute for its base classes. Here, however, extra code is needed to use the thread-safe list correctly meaning that it is not a drop-in substitute for the list it replaces. *See, e.g.*, https://en.wikipedia.org/wiki/Liskov_substitution_principle

[15] *See e.g.* https://doc.rust-lang.org/book/ch16-03-shared-state.html#using-mutexes-to-allow-access-to-data-from-one-thread-at-a-time (explaining that Rust's approach to shared mutable state attacks the "shared" problematic aspect by preventing access to the resource without first obtaining the lock); https://github.com/facebook/folly/blob/master/folly/docs/Synchronized.md (reiterating that conventional synchronization is error prone because reliant solely on conventions and explaining that Folly's Synchronized library solves this problem by inextricably linking access to the data with obtaining the lock).

[16] In pre-2011 C++ idioms, it was common to allocate objects on the heap with new and delete them with delete. These objects were often accessed by pointer. The difficulty with this was that you had to be sure the underlying object was deleted *exactly once* and, *once deleted*, never accessed again (accounting for the possibility of early

Two factors have made an isolating approach to protected resources more obtainable in C#: one very recent and one somewhat recent: disposable ref structs and Roslyn analyzers. DotNetVault uses the C# 8.0 language feature (disposable ref structs) and specifically tailored static analysis tools to isolate protected data and force locks to them to be acquired and freed in accordance with RAII.

i. C# 8's Disposable ref struct is used to isolate obtained locks on the stack and ensure prompt release in all cases

A *ref struct* is a value type that can *only* reside on the stack. Structs in general (along with *enums*) in C# are value types. Some people mistakenly believe that value types "live on the stack." This notion is deeply flawed.[17] A struct variable contains its value within itself. It can end up on the heap under two primary circumstances:

1. It is a field in a reference type
2. It is an item in a generic array (*T*[])
3. It is boxed[18]

For example, if you have a local variable of struct type and assign it to another local variable of type object (or, for example, *IDisposable*, if the struct implements that interface), then the struct is copied to the heap and a pointer to that boxed struct (and v-table) is stored in your object/*IDisposable* variable. *Ref structs*, with their counterintuitive name, are *truly impossible to store on the heap or in static memory*. They cannot be a field in a class or ordinary struct, they cannot be assigned to an object, they cannot be stored in static variables, they cannot be boxed.[19]

---

return or exceptions). Sane code thus required careful circumspection. Modern C++ eschews direct heap allocation in most instances and prefers allocating concrete objects on the stack where possible. Concrete objects on the stack are automatically cleaned up when they go out of scope. When "owning" pointers are used, the use of smart pointers such as std::unique_ptr or std::shared_ptr are preferred. These smart pointers manage deletion of the object automatically.

[17] Instead, it is better to realize that the value of a value type is stored directly in the variable (or array) that represents it. Regular structs can reside in three places: for local variables: on the stack, for struct members of reference types, on the heap, and for static variables, in static memory. For structs that are elements of an array of that struct type, the *actual value* of the struct is stored on the heap, contiguously with other elements of the array. For structs that have been boxed into a reference type, the boxed struct resides on the heap. If such a boxed struct is an element in an array of objects or interfaces, the array contains references to the elements (some or all of which may be structs or classes), not the value of the objects themselves. In other words, in *TimeSpan[]*, the values of the *TimeSpans* are stored **on the heap, contiguously, directly in the array**. In *IEquatable<TimeSpan>[]*, each element in the array is a reference to a boxed *TimeSpan* or some other object that implements this interface; only references to the elements are contiguous – the values themselves are scattered.

[18] Boxing occurs to allow structs to implement interfaces and to expose the methods common to System.Object even if it does not override them itself. Simply being stored as a struct field in a class or as an item in array of structs is not boxing. Also, storing structs in a generic collection does *not* cause boxing.

[19] https://kalapos.net/Blog/ShowPost/DotNetConceptOfTheWeek16-RefStruct; https://blogs.msdn.microsoft.com/mazhou/2018/03/02/c-7-series-part-9-ref-structs/ .

Ref structs have been with us since C# 7.2. Note that since they cannot be boxed, they also cannot implement interfaces, including IDisposable. *IDisposable* is a feature that allows, together with a using statement, RAII-like semantics to be obtained in C#. A variable declared in a using statement is guaranteed to call its *Dispose* method when its scope ends (regardless of exception or early return). With C# 8.0, you may use "using" statement (or now, declaration) with a ref struct has a void *Dispose*() method even though it cannot implement the *IDisposable* interface.

This gives us an object that cannot be stored on the heap (or static memory) in any way: it cannot be boxed, it cannot be captured in a closure, it cannot be stored in a class or ordinary struct field. It cannot be an element in an ordinary array. Its lifetime will be no longer than its scope, by inexorable guarantee. DotNetVault exploits the restricted deterministic lifetime of these types of objects to assist it in isolating a protected resource and ensuring it is properly freed in short order and not stored somewhere for an indefinite period.

ii. Static analysis prevents leakage or mingling of shared mutable state

By itself, this would not be enough to provide guarantees of isolation. Roslyn analyzers provide a framework where domain specific rules can be imposed on a program and enforced both by IntelliSense and at Build Time. DotNetVault has developed rules that enforce the isolation of resources within a vault and ensure that shared mutable state cannot leak out of a protected resource nor can external shared mutable state be accidentally mingled with it. It also can impose a rule that requires the use of a using statement or declaration by the immediate caller of a function returning a ref struct, making it a compiler error to neglect this.

In summary, DotNetVault provides the following advantages to managing shared mutable state in multithreaded applications:

1. The protected resource cannot be accessed before a lock has been obtained or after it has been released
2. The lifetime of the lock obtained cannot exceed the function in which it is called
3. The lock will be disposed of properly even in face of an exception or early return with nothing more onerous that a using declaration
4. When the lock is obtained, the user will find it very difficult to leak shared mutable state out of the lock object – most attempts will be thwarted with compiler errors: the goal is 100% isolation, but, realistically, making it very difficult to use incorrectly is probably the best that can be hoped for

5. The mechanism used is not recursive, but it will not deadlock unless a "wait forever" decision is explicitly made by the caller: by default, all attempts to obtain the lock, including in a recursive scenario, will timeout.

# 2. Prerequisites

This library and static analyzer target the C# language. It requires features that are available only in C# 8.0 and later. Generally, this will require .NET Core 3.0 or later. .NET Framework 4.8 does not, out-of-the-box, support all C# 8 features. It is possible, however, to use this library and analyzer with .NET 4.8 or .NET Standard 2.0 provided you set the language version to 8.0 manually in the *csproj* file by adding the following line in a property group that applies to all build configurations-

```
<LangVersion>8.0</LangVersion>
```

The features of C# 8.0 used by this project are "syntax-only" features[20] that work equally well with .NET Standard 2.0, .NET 4.8 and .NET Core 3.0 and perhaps other implementations. Indeed, this project was written and tested using .NET Standard 2.0 with C# 8.0 manually enabled as shown above. If using .NET Standard 2.0 or .NET Framework 4.8, you should avoid attempting to use features of C# 8.0 that require runtime/framework support.

# 3. Installation

Installation is performed by using NuGet to install the package. Officially, Visual Studio 2019 is supported on Windows; JetBrains Rider 2019.3.1+[21] is also supported. Provided a NuGet supporting extension and .NET Core 3.1+ is installed, Visual Studio Code should also work but may not provide the same level of *Intellisense* feedback for violations of the static analyzer's rules.

# 4. Usage Guide

## a. *Concept of vault-safety*

Vault-safety is a concept used extensively by this library and static analyzer. Vault-safety is a characteristic of certain types that are easy to protect. If a type is vault-

[20] https://stu.dev/csharp8-doing-unsupported-things/

[21] Rider 2019.3.1 has been tested and confirmed to work on the Amazon Linux distro. Presumably, it should also work on Mac, other Linux distros and on Windows as well.

safe, it is easier to ensure that protected data will not have references that can mutate its state or sneak out from the *Vault* barrier.

A type is vault-safe if and only if it has one or more of the following characteristics:

i. It is an *unmanaged*[22] value type – that is a value type that contains no reference types (or unsafe pointers or dynamic types) at any level of recursion in its object map. Any type that meets the *unmanaged* type constraint is *automatically* considered vault-safe.

ii. It is a concrete reference type that is *sealed* and contains only immutable data. The string type is a paradigmatic example of such a type. It is always considered vault-safe by the static analyzer. Also considered vault-safe are the collections in the *System.Collections.Immutable* namespace (except their builders), but **only to the extent that all their type arguments are vault-safe**.

iii. It is a value type that only contains fields (static and member) which themselves contain only other vault-safe types.

The reason that such types are easy to isolate is simple: assignment of those types results in either an *actual deep copy* of the object (for *unmanaged* types) or, for other vault-safe types, *effectively a deep copy*. Deep copies of data, whether true deep copies or quasi deep copies due to immutability, cannot be used to change the value protected. Concerns about references to the protected data sneaking out of the ambit of their protective isolation do not apply to these types.

*Unmanaged* types require no special annotation to be considered vault-safe: they are manifestly vault-safe by their very nature. For reference types and value types that do not comply with the *unmanaged* constraint, you must either annotate the type with the *VaultSafeAttribute*[23] or, alternatively place its type in the whitelist file[24]. It is recommended that the *VaultSafeAttribute* be employed for types over which you have control and the whitelist for types beyond your ability to annotate with attributes (for example, because they are part of a 3rd-party API).

---

[22] "*unmanaged*" is a type constraint introduced in C# 7.3. According to the standard, a type is unmanaged if "is not a reference type and doesn't contain reference type fields at any level of nesting." https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-7.3/blittable . Types parameters that meet the requirements of this constraint can have their address assigned to a pointer inside *unsafe* generic methods.

[23] v. §§ 6.a, 5.b, *infra*.

[24] *Id*.

b. *Overview of Tools*

As a consumer of this project, there are two broad categories of types provided for your use. *Vaults* isolate protected resources and prevent concurrent access to them or any values in their object graph. To obtain temporary access to an object protected by a *Vault* you call the one of the *Vault's Lock()* or *SpinLock()* methods to obtain the second category of type: the *LockedResource*. The *LockedResource* is an RAII type that may only be stored on the stack and must be disposed by the same Method (or Property) that obtained it. In fact, the method or property that obtains it must guard it with a *using* statement or declaration. Failing to do so is a compiler error.

i. *Vaults*

Vault objects all inherit from the abstract base class *Vault<T>*. There are two axes of types of vaults. One axis represents the underlying synchronization mechanism employed by the vault (atomic, monitor lock and ReaderWriterLockSlim[25]). The other axis represents the type of resource protected by the vault. There are two Vaults provided for your use out-of-the-box: the *MutableResourceVault<T>* and the *BasicVault<T>*. The difference between which should be used depends on whether *T* is vault-safe.[26] The *BasicVault<T>* should be used to protect vault-safe resources because the *LockedResource* provided by the *BasicVault<T>* is minimally restrictive. The *MutableResourceVault<T>* should be used to protect resources that are not vault-safe: the *LockedResource* object it provides when you obtain a locked resource prevents references to the protected data from leaking out and prevents non-vault safe external data from creeping in.[27] Finally an abstract *CustomizableMutableResourceVault<T>*, inherits from *MutableResourceVault<T>* and exits to allow you to make your own version of the *MutableResourceVault<T>* with your own *LockedResource* type to provide an accessible API for frequently used operations on a type that is not vault-safe.

ii. *LockedResources*

Every publicly accessible *Vault* object contains public methods called *Lock* and *SpinLock*. These methods, all of which are annotated with the *UsingMandatory* attribute,

[25] v. ReaderWriterLockSlim Class
[26] *See* Concept of vault-safety, *supra* at a.
[27] You *may* use a *MutableResourceVault* to protect a vault-safe object as well. It may be advisable to use it if the resource you are protecting is under active development and might lose its vault-safe status later. If the resource is unlikely to need to become non-vault-safe, the BasicVault should be used because its locked resource is less restrictive and easier to work with.

"check-out" the protected resource give you temporary access to the resource mediated by a *LockedResource* object. *LockedResource* objects are Disposable *ref structs*[28] that grant you limited access to the resource and automatically return the resource to the owning *Vault* when their lifetime ends. BasicVault<T> provides a *LockedResource* of type *LockedVaultObject<TVault, T>* and *MutableResourceVault<T>* provides a *LockedResource* of type *LockedVaultMutableResource<TVault, TResource>*. *LockedVaultObject<TVault, T>* simply exposes the protected resource to you as a read/write Property: this is all the protection a vault-safe resource requires. The *LockedMutableResourceVault<TVault, TResource>* restricts your interaction with the protected resource to the use of certain specially annotated delegates. The annotations on these delegates cause the static analyzer to refuse to compile your code if it cannot prove the code you passed in the delegate will not leak mutable state out of the resource or permit externally accessible mutable state from becoming part of the resource.



[28] See § 1.c.i , *supra*.

### *c. Vaults In-Depth*

#### i. Underlying Synchronization Mechanisms

The vaults provided supply a variety of underlying synchronization methods for you to choose from. Also, they all share a common API, enabling you to easily change your chosen synchronization mechanism at compile time.[29] The available synchronization mechanisms are atomics, standard monitor lock and ReaderWriterLockSlim.

#### ii. Atomic Vaults

The atomic vaults *(BasicVault<T>*, *MutableResourceVault<T>* and *CustomizableMutableResourceVault<T>*) are all derived from the *AtomicVault*<T> abstract base class. Prior to the introduction of version 0.2.x of this project, all vaults were atomic vaults. Synchronization is obtained through use of interlocked exchange methods that attempt to retrieve a reference to the protected resource from within the vault; this method suffices to ensure synchronized access to the protected resource. The *Lock* and *SpinLock* methods[30] obtain the locks using different strategies. *Lock* will sleep for a short period between failed attempts, *SpinLock* is a busy wait.[31]

#### iii. Monitor Vaults

Monitor vaults (*BasicMonitorVault<T>*, *MutableResourceMonitorVault<T>* and *CustomizableMonitorMutableResourceVault<T>*), use the standard .NET Monitor methods with a synchronization object to achieve synchronization.[32] It offers significant advantages over use of those facilities directly:

i. RAII (like the *lock* syntax) is used *in all forms* not solely untimed acquisition attempts.

---

[29] This ability is a *compile-time* ability. The use of dynamic polymorphism was attempted but altered performance of the tests sufficiently that it was abandoned. Many multithreaded applications are sensitive to performance and avoidance of unnecessary indirection is more important than providing the ability to change synchronization mechanisms *at runtime*.

[30] v. insert reference infra.

[31] Because all vaults in prior versions of the software were based on atomics, other vaults (with different synchronization) have both versions of these methods as well. In these other vaults, however, *Lock* and *SpinLock* are identical.

[32] v. Monitor Overview

ii. Time-limited attempts to obtain the lock are the default: it is easier to find the cause of a *TimeoutException* than it is to reproduce and identify a deadlock.
iii. It offers the ability to use a cancellation token to propagate a request to cancel the attempt to obtain the lock (by itself or in addition to the usage of a timeout period)
iv. It prevents access to the protected resource whenever the lock is not obtained
v. It does not permit recursive acquisition of locks, eliminating a bad coding practice that is – at any rate – totally unnecessary when you are protecting a resource with a vault.

The Monitor Vault, unlike the Atomic Vault, does allow for you to simply obtain the resource without using a Timeout or Cancellation token. To do this, however, you must use the ominously named *LockBlockUntilAcquired* method to explicitly request this behavior. Generally, until you encounter a scenario where the small overhead for a timed acquisition becomes problematic in a performance sensitive use-case, you should simply use the Lock method instead (which uses a default timeout if you do not specify one). You can always switch to *LockBlockUntilAcquired* after you have satisfied yourself that there will be no deadlocks and you need to avoid the overhead of the timed acquisition.

iv. ReadWrite Vaults

These vaults (currently only a *BasicReadWriteVault<[VaultSafeTypeParam] T>*) use *ReaderWriterLockSlim*[33] as their synchronization mechanism. Unlike the other vaults, it provides multiple modes of access to the protected resource object. It offers shared access across multiple threads to the protected resource when acquired in read-only mode and exclusive access to the protected resource when acquired in read-write mode. It also provides an *upgradable* read-only mode: one thread at a time may acquire shared read-only access to the resource in upgradable mode – while holding such an upgradable read-only lock, it either release the lock when complete or choose to acquire a read-write lock *without needing first to release its read-only* lock. This software always uses this mechanism with a no-recursion policy. Usage of this software provides the following advantages over using *ReaderWriterLockSlim* directly:

i. RAII: no possibility of causing a deadlock from forgetting to release a lock or for releasing them in the wrong order. When your lock goes out of scope, it is released.

[33] v. 25, *supra*.

ii. Time-limited attempts to obtain the locks are the default in all modes: it is easier to find the cause of a *TimeoutException* than it is to reproduce and identify a deadlock.
iii. It offers the ability to use a cancellation token to propagate a request to cancel the attempt to obtain the lock (by itself or in addition to the usage of a timeout period)
iv. It prevents access to the protected resource whenever the lock is not obtained
v. It does not permit recursive acquisition of locks, eliminating a bad coding practice that is – at any rate – totally unnecessary when you are protecting a resource with a vault.

ReadWrite locked resource objects are somewhat unique: the upgradable readonly locked resource object provides *the same Lock and SpinLock* methods the vault does to enable you to upgrade your lock to a read-write lock. At present, read-write vaults only support VaultSafe resources because it is easier to prevent write access to these types. Adding specific read-write vaults for common collection types and adding a version to support protection of resources that are not vault-safe are currently under consideration.

v. Functionality Common to All Vaults (intended for public consumption)

1) Public Read Only Properties:

| Type | Name | Description |
|---|---|---|
| bool | DisposeInProgress | The dispose operation is currently in progress |
| bool | IsDisposed | The vault has been disposed. |
| TimeSpan | DefaultTimeout | The default maximum amount of time to spend attempting to obtain a lock. |
| TimeSpan | DisposeTimeout | How long to wait for a locked resource to be returned to the vault when the Dispose method is called. If the locked resource is not returned to the vault in time, an exception will be thrown. |
| TimeSpan | SleepInterval | ATOMIC VAULTS: How long should the thread sleep between failed attempts to obtain a lock. Typically, a few milliseconds. Not used at all when *SpinLock* is called. |
| | | OTHER VAULTS: how long should we wait to obtain the resource before checking the *CancellationToken* to see if cancellation has been requested. |

*Figure 6* – Public Properties Common to All Vaults

*2) Dispose* and *TryDispose*

| Type | Name | Params | Description |
|---|---|---|---|
| void | *Dispose* | | Disposes the current vault. If the protected resource is *IDisposable*, it is disposed. Note that this method will throw an exception if it cannot obtain a lock in the time specified by the *DisposeTimeout* property. You should be sure that all threads that might be obtaining and releasing locks have stopped before calling. Alternatively, use *TryDispose*. |
| bool | *TryDispose* | [timeout : TimeSpan] | Try to dispose the vault for up to the time specified by timeout. Returns true for success, false for failure (indicating unable to obtain lock in time specified). |

*Figure 7* – Public Methods Common to All Vaults

3) Lock Acquisition Method Groups

*i. Lock* and *SpinLock*

The *Lock* method group is common across all types of vaults: it means attempt to obtain an exclusive read/write lock. For Atomic Vaults, it will sleep between failed attempts in contrast to *SpinLock*. *SpinLock*, for Atomic Vaults, busy-waits to obtain the exclusive read/write lock; **for all other vaults, *SpinLock* is identical to *Lock***: it is included to facilitate ease of switching between Monitor and Atomic vaults only.

*ii. LockBlockUntilAcquired*

*LockBlockUntilAcquired* is available for Monitor and ReadWrite vaults. Unlike *Lock* and *SpinLock*, which, when called with no parameters, use a timeout, *LockBlockUntilAcquired* (and its cousins *RoLockBlockUntilAcquired* and *UpgradableRoLockBlockUntilAcquired*) potentially can cause deadlocks. It is recommended that you use *Lock* with the default timeout instead to help you identify potential sources of deadlocks. If you are confident that there will be no deadlocks (you always acquire multiple locks in the same order) and the overhead from using a timed version becomes a bottleneck, switching to *LockBlockUntilAcquired* may improve performance. It is unavailable for Atomic Vaults.

*iii. RoLock, UpgradableRoLock*

These method groups are available only with ReadWrite vaults. Unlike *Lock* and *SpinLock*, these methods produce non-exclusive but read-only locked resource objects. An indefinite number of threads may simultaneously hold these non-exclusive read-only locks, except that only one thread at a time may hold an *upgradable* read-only lock. No thread may hold a read-only lock while *any* thread holds an exclusive read-write lock. For example, if three threads hold read-only locks, a fourth thread may also acquire a read-only lock or an upgradable read-only lock. If any one of the three threads had held an upgradable read-only lock, the fourth thread could not obtain the upgradable read-only lock until the thread holding it released it. Write locks are exclusive: no thread can obtain a write lock while any thread holds any other type of lock.

Upgradable locks are unique: if a thread holds an upgradable read-only lock, it may attempt to obtain an exclusive write lock without releasing its own read-only lock. If all other threads release their locks before timeout or cancellation, the lock will be upgraded to an exclusive read-write lock. When the read-write lock is released, the

thread will still hold an upgradable read-only lock until it releases that lock as well. The underlying mechanism for this vault is ReaderWriterLockSlim.[34]

*iv. RoLockBlockUntilAcquired, UpgradableRoLockBlockUntilAcquired*

These method groups are available only with ReadWrite vaults. Unlike *RoLock* and *UpgradableRoLock*, which, when called with no parameters, use a timeout, *RoLockBlockUntilAcquired* and *UpgradableRoLockBlockUntilAcquired* can potentially deadlock. It is recommended that you use *RoLock* or *UpgradableRoLock* with the default timeout instead to help you identify potential sources of deadlocks. If you are confident that there will be no deadlocks (you always acquire multiple locks in the same order) and the overhead from using a timed version becomes a bottleneck, switching to *RoLockBlockUntilAcquired* or *UpgradableRoLockBlockUntilAcquired* may improve performance.



[34] v. 25, *supra*.

### *v. Summary of Method Groups*

*Table 1* – Summary of Lock Acquisition Method Groups by Vault Type

| Lock Acquisition Methods Summary | | | | |
|---|---|---|---|---|
| **Method Group** | **Atomic Vaults** | **Monitor Vaults** | **Read Write Vaults** | **Upgradeable Locked Resource Objects** |
| **Lock** | Loop until the resource is acquired; Sleep a bit between failed attempts at acquisition. | Use Monitor.TryEnter or Enter to acquire the lock; Periodically wake up to check for timeout or cancellation request | Use ReaderWriterLockSlim.TryEnterWriteLock to acquire an exclusive read-write lock; Periodically wake up to check for timeout or cancellation request | Use ReaderWriterLockSlim.TryEnterWriteLock to acquire an exclusive read-write lock; Periodically wake up to check for timeout or cancellation request |
| **SpinLock** | Loop until the resource is acquired; Do not sleep; Busy wait. | Use Monitor.TryEnter or Enter to acquire the lock; Periodically wake up to check or timeout or cancellation request | Use ReaderWriterLockSlim.TryEnterWriteLock to acquire an exclusive read-write lock; Periodically wake up to check for timeout or cancellation request | **N/A** |
| **LockBlockUntilAcquired** | **N/A** | Use Monitor.Enter to acquire the lock; potentially deadlock | Use ReaderWriterLockSlim.EnterWriteLock to acquire an exclusive read-write lock; potentially deadlock | Use ReaderWriterLockSlim.EnterWriteLock to acquire an exclusive read-write lock; potentially deadlock |
| **RoLock** | **N/A** | **N/A** | Use ReaderWriterLockSlim.TryEnterReadLock to acquire a non-exclusive read-only lock; Periodically wake up to check for timeout or cancellation request | **N/A** |
| **RoLockBlockUntilAcquired** | **N/A** | **N/A** | Use ReaderWriterLockSlim.EnterReadLock to acquire a non-exclusive read-only lock; Periodically wake up to check for timeout or cancellation request | **N/A** |
| **UpgradableRoLock** | **N/A** | **N/A** | Use ReaderWriterLockSlim.TryEnterUpgradableReadLock to acquire an a non-exclusive read-only lock (only one read-only lock at a time can be upgradable so it is exclusive vis-à-vis other upgradable read-only locks but non-exclusive vis-à-vis read-only locks in general; Periodically wake up to check for timeout or cancellation request | **N/A** |
| **UpgradableRoLockBlockUntilAcquired** | **N/A** | **N/A** | Use ReaderWriterLockSlim.EnterUpgradableReadLock to acquire an a non-exclusive read-only lock (only one read-only lock at a time can be upgradable so it is exclusive vis-à-vis other upgradable read-only locks but non-exclusive vis-à-vis read-only locks in general; Potentially deadlock | **N/A** |

*4) Method Group Overloads*

The overloads of these methods are summarized below:

| **Overload Summaries for Acquisition Methods other than Block Until Acquired** | | |
|---|---|---|
| **Overload Params** | **Description** | **Throws** |
| Parameterless | Attempt to obtain resource for the vault's default timeout period. | *ObjectDisposedException; TimeoutException; LockAlreadyHeldThreadException*[35] |
| *TimeSpan* | Attempt to obtain resource for period specified by *TimeSpan* parameter | *ObjectDisposedException; ArgumentOutOfRangeException; TimeoutException; LockAlreadyHeldThreadException*[36] |
| CancellationToken | Attempt to obtain resource until successful, or a cancelation request is propagated to the CancellationToken | *ObjectDisposedException; OperationCanceledException; LockAlreadyHeldThreadException*[37] |
| *TimeSpan; CancellationToken* | Attempt to obtain until the earliest of: a- successfully obtaining resource, b- period specified by *TimeSpan* parameter exceeded, c- cancellation request propagated to *CancellationToken* parameter | *ObjectDisposedException; ArgumentOutOfRangeException; TimeoutException; OperationCanceledException; LockAlreadyHeldThreadException*[38] |

*Figure 8 – Lock* and *Spinlock* Overloads

vi. *Basic Vaults*

Basic Vaults implement the *IBasicVault*<[VaultSafeTypeParam] T> interface, include *BasicVault*<*T*> (an Atomic Vault), *BasicMonitorVault*<T> (a Monitor Vault) and BasicReadWriteVault<T> (a ReadWrite vault). All Basic Vaults are easy to use but come with a significant limitation: they can only protect vault-safe resources. You will thus be limited to certain classes of value types and immutable reference types. If you

[35] Atomic vaults do not throw this exception. They will timeout unless a *CancellationToken* is supplied but no *TimeSpan*. If supplied a token but no timespan they will either deadlock or, if cancellation is requested, throw *OperationCanceledException*.
[36] *Id*.
[37] *Id*.
[38] *Id*.

acquaint yourself with the idioms of programming with immutable data structures, this is not difficult: C#'s immutable collections[39] have highly convenient methods that can create new collections with different values based on their current contents. Also, since large mutable structs can now be easily accessed by reference and read-only reference, large mutable structs make an excellent option for a "State Flags" type object used to manage a multithreaded state machine. Aside from large mutable structs, Basic Vaults protect resources such as *DateTime*, *string*, *ulong*, as well as immutable collections of such values.

Because the BasicVault protects values that are intrinsically vault-safe, it can supply convenience functions that other vaults cannot provide. These functions are declared in *IBasicVault<[VaultSafeTypeParam] T>* and include:

- `T CopyCurrentValue(TimeSpan timeout)`[40]
- `(T value, bool success) TryCopyCurrentValue(TimeSpan timeout)`[41]
- `void SetCurrentValue(TimeSpan timeout, T newValue)`[42]
- `bool TrySetNewValue(TimeSpan timeout, T newValue)`[43]

### vii. Mutable Resource Vaults

These vaults can protect resources that are **not** vault safe and include *MutableResourceVault<T>* (an atomics-based vault) and *MutableResourceMonitorVault<T>* (a monitor-based vault).[44] These vaults are designed to protect mutable resources and are far more restrictive than Basic Vaults. The key to understanding **provably** *thread-safe interaction* with a mutable resource is to realize that all such interactions with the mutable resource must only read from or write to vault-safe objects (except for the protected resource itself). Consider updating a *StringBuilder* object, a paradigmatic example of a resource that is mutable and not vault-safe. **The**

---

[39] These are contained in the *System.Collections.Immutable* namespace. They are included as part of .NET Core and are available as a NuGet package for the .NET framework. DotNetVault's static analyzer (and Roslyn analyzers generally) make extensive use of them to allow concurrent execution of analysis without fear of corrupting mutable state. All multithreaded projects in C# should make heavy use of them.

[40] Copies the current value protected by the vault. Will throw *ArgumentOutOfRangeException, TimeoutException and ObjectDisposedException.* Returns a copy of the protected resource. If the vault is a ReadWrite vault, a read-only lock is obtained (and released) internally during the execution of this method.

[41] Tries to copy the value currently protected by the vault. Will return (true, value) on success or (false, undefined) on timeout. Will throw *ObjectDisposedException* and *ArgumentOutOfRangeException*. If the vault is a ReadWrite vault, a read-only lock is obtained (and released) internally during the execution of this method.

[42] Replaces the value currently protected by the vault with a specified new value. Will throw *ArgumentOutOfRangeException*, *TimeoutException* and *ObjectDisposedException*.

[43] Tries to replace the value currently protected by the vault with a specified new value. Will return true for success and false for a timeout failure. Will throw *ArgumentOutOfRangeException* and *ObjectDisposedException*.

[44] No read-write vaults are currently provided for resources that are not vault safe. I plan to explore options around mutability analysis, providing options for collections of vault safe types (where the collections need not be vault safe) or introducing a read-only wrapper system where client code will be responsible for ensuring that the wrapper performs no externally visible write operations.

**data used to update the builder must itself be vault-safe.**[45] Otherwise, mutable state accessible from outside the vault could cause a change in the state of the protected resource. This is the precise area where a reference-based, garbage collected languages like C# are at cross-purposes with thread-safety.[46]

### 1. *Factory Methods*

Special care must be taken when constructing a mutable resource vault. You must ensure that **only the mutable resource vault ever can see a reference to the value it protects**. To facilitate this, the factories for mutable resource vaults require a delegate of type *Func<T>* as well as a positive *TimeSpan* defining the *DefaultTimeout* for the vault. You must ensure that the object constructed by the *Func<T>* -- as well as any non-vault-safe subparts of it are not accessible anywhere outside the delegate. The following example code shows correct and incorrect ways to accomplish this:

```
internal static class MutableResourceVaultCreationExamples
{
        internal static void CreateMutableResourceVaultTheCorrectWay()
        {
            var sbVault =
                MutableResourceVault<StringBuilder>.CreateMutableResourceVault(() =>
                        new StringBuilder(), TimeSpan.FromMilliseconds(250));
            Console.WriteLine(@$"We just created a [{sbVault}] in the correct way -- " +
                              @"only the mutable resource vault will ever see the" +
                                        "StringBuilder it constructs.");
        }
        internal static void CreateMutableResourceVaultTheIncorrectWay()
        {
            var sbVault =
                MutableResourceVault<StringBuilder>.CreateMutableResourceVault(() =>
                        BadPreExistingStringBuilder,
                        TimeSpan.FromMilliseconds(250));
            Console.WriteLine(
                $@"We just created an [{sbVault}] in a very, very bad way.  "+
                @"The vault now protects a resource that pre-existed the vault."+
                @$"  A change to {nameof(BadPreExistingStringBuilder)} is not threadsafe"
                + @"and will propagate to the vault's protected resource!");
        }

        private static readonly StringBuilder BadPreExistingStringBuilder = new
                StringBuilder();
}
```

*Figure 9* – Correct and Incorrect Creation of *MutableResourceVault*

[45] One would not pass a *StringBuilder* or a *List<char>* to append to the end of the protected string builder. Instead one would pass a *string* (vault-safe) or an *ImmutableArray<char>* or *ImmutableList<char>* to the *StringBuilder*. In this way we can be sure that no subsequent external change to the objects referred to by these parameters will not affect the protected resource.

[46] In a language like C++ or, especially, Rust where value semantics are used ubiquitously and objects can be, frequently are, and should be – wherever possible – on the stack, there is little danger of concurrent access. When passed by value, they are deep copies; when moved, the resource is no longer accessible from the original variable. In C++ and *unsafe* blocks in Rust this can be defeated by pointers, references and reference-like view objects (e.g. std::string_view and std::span). In C#, where most objects, by design, are on the heap and accessed indirectly, mutable state almost begs to be shared.

Care must also be taken to ensure that any non-vault-safe sub-objects of the constructed protected resource are not visible outside of the vault. The following example shows slightly more subtle correct and incorrect construction of a *MutableResourceVault*:

```
internal static class SlightlyMoreSubtleExample
{
    internal static void CreateMoreComplicatedMutableResourceTheCorrectWay()
    {
        var sbVault = MutableResourceVault<PrettyBadMoreComplexExample>
            .CreateMutableResourceVault(() =>

            {
                var sbOne = new StringBuilder();
                var sbTwo = new StringBuilder();
                return new PrettyBadMoreComplexExample(sbOne, sbTwo);
            }, TimeSpan.FromMilliseconds(250));

        Console.WriteLine(@$"I just created a more complicated {sbVault} in the correct way." +
                          @"  Neither the PrettyBadMoreComplexExample nor any of its subobjects are " +
                          @"accessible outside the mutable resource vault.");
    }

    internal static void CreateMoreComplicatedMutableResourceInASlightlySubtleIncorrectWay(
        StringBuilder shouldBeSecond)
    {
        var sbVault =
            MutableResourceVault<PrettyBadMoreComplexExample>.CreateMutableResourceVault(() =>
                {
                    var sbOne = new StringBuilder();
                    //VERY BAD!  Any change to ShouldBeSecond (which is accessible outside the vault)
                    //is not thread-safe
                    //and will propagate to the value in the vault!
                    return new PrettyBadMoreComplexExample(sbOne, shouldBeSecond);
                },
                TimeSpan.FromMilliseconds(250));
        Console.WriteLine(@$"I just created a {sbVault} in a very unsafe but subtle way.  " +
                          @$"If anyone changes the object referred to by {nameof(shouldBeSecond)}," +
                          @"It will propagate in an unsafe way to the value protected by the vault.");
    }
}

internal sealed class PrettyBadMoreComplexExample
{
    public StringBuilder FirstBuilder { get; set; }
    public StringBuilder SecondBuilder { get; set; }

    internal PrettyBadMoreComplexExample(StringBuilder first,
        StringBuilder second)
    {
        FirstBuilder = first;
        SecondBuilder = second;
    }
    internal PrettyBadMoreComplexExample()
    {
        FirstBuilder = new StringBuilder();
        SecondBuilder = new StringBuilder();
    }
}
```

*Figure 10* – More Elaborate Correct and Incorrect Creation of *MutableResourceVault*

After correctly constructing a Mutable Resource Vault such that neither it nor any non-vault-safe sub-object thereof are accessible from outside the mutable resource vault,

the static analysis rules enable a high degree[47] of confidence that no externally accessible mutable state will be mingled with the protected resource in the vault nor will any such mutable state be leaked out of the vault. The responsibility for ***constructing*** *the object in a way that does not expose it or any non-vault-safe subpart thereof* lies solely with the user.

*2. Public Methods and Properties*

Except for lacking the convenience methods defined in *IBasicVault<T>*,[48] The methods and properties exposed by mutable resource vaults are like those exposed by basic vaults.[49] The difference in usage lies not in the vaults itself, but in the functionality and flexibility of the locked resource objects they supply when a lock is obtained. Mutable resource vaults are available in atomic and monitor varieties.

viii. *CustomizableMutableResourceVault<T>*

As shown below, the usage of the *MutableResourceVault*'s[50] *LockedResource* lends itself to a manageable yet somewhat awkward and inconvenient syntax. For that reason, DotNetVault provides the *CustomizableMutableResourceVault* and examples showing how to create your own Vault Objects with their own *LockedResources* to which you can supply common operations in a more convenient syntax. The project itself provides a well-documented classes called *StringBuilderMonitorVault* (monitor-based) and *StringBuilderVault* (using atomics) and their *LockedResource* objects called *LockedMonitorStringBuilder* (monitor) and *LockedStringBuilder* (atomic) showing how to make similar customized for your own classes and provide a very easy-to-use wrapper around the *MutableResourceVault*. It is hoped, as a future feature, to add code generation facilities to remove the boilerplate aspect of making these. That said, it is quite possible to use *MutableResourceVault* on its own and designing your own custom Vaults and *LockedResource* objects is a largely mechanical and easy, though tedious, task. If a vault protecting a non-vault-safe resource is to be used extensively, it is highly recommended that you take the time to customize the vault that protects it. Extension methods defined on the locked resource object also provide an option for improved readability.[51]

---

[47] Although I have performed testing and believe that it will be difficult to mingle mutable state with the protected object or to allow mutable state to leak from the vault, I cannot guarantee it. This project is released, under the MIT license, with no warranties, not even the warranty of merchantability or fitness for any particular purpose. See the license agreement for more legal details. I welcome any reports of ways found that circumvent the protection afforded by this project's static analyzer and welcome proposed fixes and new static analysis rules to make the protections provided by this project stronger.

[48] v. § 4.c.6, *supra.*

[49] v. § 4.c.3, *supra.*

[50] Of either the monitor or atomic varieties.

[51] v. Figure 21, *infra.*

### *d. LockedResources In-Depth*

#### *i.* Common Functionality

All *LockedResources* are *ref structs*[52] that can only be located on the stack. Such objects cannot be boxed, stored in static memory or made a field of any type that is not itself a *ref struct*. Each has a Dispose method that returns the resource it guards to that vault from which it was obtained. When released for public consumption, the method returning these objects is always annotated with the *UsingMandatoryAttribute* which mandates that the immediate caller of the method immediately guard the return value with a *using* statement or declaration. This rule prevents failure to promptly Dispose the *LockedResource*, which is *always* a bug.

#### *ii.* Vaults and their LockedResources

Aside from any custom locked resource objects provided or which you may create, there are three broad categories of vaults with someone distinct functionality. These categories include Basic Vaults (whether of the atomic or monitor varieties), Mutable Resource Vaults (whether of the atomic or monitor varieties) and ReadWrite vaults (which presently support only vault safe resources and are analogous to Basic Vaults in that respect). The following chart describes the locked resource objects and their functionality.



[52] v. #19, *supra*.

*Table 2* -- Categories of Locked Resource Objects and Their Characteristics

| Categories of Locked Resource Objects and Their Characteristics | | |
|---|---|---|
| **Category** | **Vault Type / Its Locked Resource Object** | **Distinctive Characteristics** |
| Basic Vaults | BasicVault<T> / LockedVaultObject<BasicVault<T>, T><br>_____<br>BasicMonitorVault<T> /<br>LockedMonVaultObject<BasicMonitorVault<T>, T> | These locked resource objects expose their locked resource object by writable reference called Value. They are, by far the most straight-forward to use of the locked resource objects but are limited to only protecting vault safe resources. |
| Mutable Resource Vaults | MutableResourceVault<T> / LockedVaultMutableResource<<br>MutableResourceVault<T>, T><br>_____<br>MutableResourceMonitorVault<T> /<br>LockedMonVaultMutableResource<<br>MutableResourceMonitorVault<T>, T> | These locked resources require access to their protected resources be through special delegates annotated with attributes that prevent protected mutable state from the protected resource leaking to the outside or mutable state from the outside becoming mingled with the protected resource. |
| Basic ReadWrite Vaults | BasicReadWriteVault<T> /<br>(multiple, detailed in separate table) | These vaults allow varying levels of access to their protected resource with varying levels of exclusivity. The read-only version of the locked resource object can be obtained on multiple threads simultaneously. The protected resource is, as with the BasicVault, exposed as a property called value. For read-only locked resources, this value is returned by read-only reference. The writable version of the locked resource is exclusive and returned by (writable) reference. There is also an upgradable readonly locked resource object -- but only one thread may hold an *upgradable* locked resource at a time (although other threads may simultaneously hold a normal read-only locked resource) |

*Table 3* ReadWrite Vault Locked Resource Objects and Their Characteristics

| Categories of ReadWrite Vault's Locked Resource Objects and Their Characteristics | | |
|---|---|---|
| **Lock Access Level** | **Locked Resource Object Type** | **Distinctive Characteristics** |
| Read-Only | ReadOnlyRwLockedResource<<br>BasicReadWriteVault<T>, T> | Many threads can hold simultaneously. Protected resource is exposed by read-only reference to prevent mutation. |
| Upgradable Read-Only | ReadOnlyUpgradableRwLockedResource<<br>BasicReadWriteVault<T>, T> | One of (potentially many) threads may hold an *upgradable* read-only locked resource at a time. (Other threads may simultaneously hold a non-upgradable read-only locked resource object.) This object is unique in that it can be used to obtain a writable lock *without* releasing its read-only lock first. It exposes Lock methods like those found in vaults. |
| Writable | RwLockedResource<<br>BasicReadWriteVault<T>, T> | If any thread holds a writable locked resource object, no other thread may at that time hold any other type of locked resource object from the same vault. It is an exclusive lock that (unlike read-only varieties) exposes its protected resource by writable reference |

### *iii.* Suggestion Regarding Declaration of Locked Resource Objects (i.e., Use "*var*")

Vaults and locked resource objects make every effort – to the extent possible -- to expose a similar API. Since dynamic polymorphism is not possible for Locked Resource objects, the similar API allows for *compile-time* compatibility only: if you are using a BasicMonitorVault and want to test out the performance characteristics of the atomic version, you should simply be able to swap out the vault object. For that reason, it is *highly recommended* that you use type inference when declaring locked resource objects. The names of the types for these objects are long and annoying to type; additionally, if you declare them explicitly, when you swap out one type of vault for another, you will have to update every locked resource declaration that uses that vault. If you cannot use type inference via "var" (perhaps because your organization's coding standards frowns upon it) an alternative is to alias the vault's locked resource object with *using* at the top of every file that references the vault. Thus, if you swap the vault, you will have to update only one alias per file rather than update every single declaration. A better solution is to convince your organization to abandon its reluctance to embrace type inference – or, at least, to selective abandon it when it comes to using this library.

*iv.* Locked Resource Objects of Basic Vaults
( *LockedVaultObject<BasicVault<T>, T>* and *LockedMonVaultObject<BasicMonitorVault<T>, T>* )

These objects are associated with Basic Vaults.[53] *TVault* denotes the owning vault's type; *T* is the type of protected resource. The *VaultSafeTypeParamAttribute* informs the static analyzer that all type arguments matched with this parameter must be vault-safe. The property "*Value*" exposes the protected resource and returns it by reference.

A static analysis rule prevents storing an alias to the property into a ref local, which could enable access to the resource after the lock is released.[54] Since its type must be vault-safe, one need not worry about mingling it with unprotected resources or retaining a copy: if it is a reference type, the object to which it refers is immutable; if a value type, it is a deep copy or *effectively* a deep copy.[55] Accessing the property of the lock object directly, because it is by reference, will propagate changes to the protected resource. If you copy the protected resource into a local, you will have to reassign it to the lock's Value property if you want the change to the copy to be reflected in the protected resource. The following code sample demonstrates the need for the rule as well as how return-by-reference works:

[53] v. § 4.c.ii, *supra*.
[54] v. §§ 5.h, 6.i, *infra*.
[55] v. § 4.a, *supra*.

```
internal static void LargeMutableStructsAreNowEfficientProtectedResourcesWithReturnByReference()
{
    using var lck = TheVault.SpinLock();
    Console.WriteLine($"Printing protected resource:\t\t{lck.Value.ToString()}");
    var copy = lck.Value;
    Console.WriteLine($"Printing copy of protected resource:\t\t{copy.ToString()}");
    Console.WriteLine("Incrementing protected resources count and appending text.");
    lck.Value.Count += 1;
    lck.Value.Text += " Accessing large structs by reference is efficient!";
    Console.WriteLine(lck.Value);
    Console.WriteLine($"Printing protected resource:\t\t{lck.Value.ToString()}");
    Console.WriteLine($"Printing unaffected copy of protected resource:\t\t{copy.ToString()}");
    Console.WriteLine("Going to change the value of the copy so count is 12 and text is `in lecto sunt tuo!'.  This will
         not affect protected resource.");
    copy.Text = "in lecto sunt tuo!";
    copy.Count = 12;
    Console.WriteLine($"Printing protected resource:\t\t{lck.Value.ToString()}");
    Console.WriteLine($"Printing independently mutated copy of protected resource:\t\t{copy.ToString()}");
    Console.WriteLine("We can, of course, work on the copy and then overwrite the protected resource with it.  " +
                      "For large structs, it is probably best to just work through the .Value property.");
    lck.Value = copy;
    Console.WriteLine("now protected resource and copy will once again be the same...");
    Console.WriteLine($"Printing protected resource:\t\t{lck.Value.ToString()}");
    Console.WriteLine($"Printing copy of protected resource:\t\t{copy.ToString()}");

    Console.WriteLine($"{nameof(LargeMutableStructsAreNowEfficientProtectedResourcesWithReturnByReference)} functions
         properly.");
}

internal static void DemonstrateNeedForRuleAgainstLocalRefAlias()
{
    Console.WriteLine($"Starting {nameof(DemonstrateNeedForRuleAgainstLocalRefAlias)}.");
    try
    {
        MyMutableStruct value = MyMutableStruct.CreateMutableStruct(DateTime.Now, 42, "Hello there!");
        ref MyMutableStruct alias = ref value;
        {
            using var lck = TheVault.SpinLock();
            //following line will not compile because it may permit unsynchronized access after lck's
            //lifetime ends.  If you want to access by reference, simply use lck.Value and changes will propagate
            //efficiently.
            alias = ref lck.Value;
        }
        //The following line will result in unsynchronized access to mutable resource.
        //for this reason, aliasing the protected resource into a ref local is forbidden.
        alias.Count *= 42;
        Console.WriteLine(TheVault.CopyCurrentValue(TimeSpan.FromMilliseconds(250)));
    }
    catch (Exception ex)
    {
        Console.Error.WriteLine($"{nameof(DemonstrateNeedForRuleAgainstLocalRefAlias)} FAILED.  Exception: [{ex}]." );
        throw;
    }

}
private static readonly BasicVault<MyMutableStruct> TheVault =
    new BasicVault<MyMutableStruct>(MyMutableStruct.CreateMutableStruct(DateTime.Now, 0, "Hello, DotNetVault!"));
```

*Figure 11 -- Return by Reference and Ref Local Alias Prohibition*

OUTPUT:

```
Printing protected resource: [MyMutableStruct] -- Time: [2020-02-13T13:34:31.9985334-05:00], Count: [0], Text: [Hello,
        DotNetVault!].
Printing copy of protected resource: [MyMutableStruct] -- Time: [2020-02-13T13:34:31.9985334-05:00], Count: [0], Text:
        [Hello, DotNetVault!].
Incrementing protected resources count and appending text.
[MyMutableStruct] -- Time: [2020-02-13T13:34:31.9985334-05:00], Count: [1], Text: [Hello, DotNetVault! Accessing large
        structs by reference is efficient!].
Printing protected resource: [MyMutableStruct] -- Time: [2020-02-13T13:34:31.9985334-05:00], Count: [1], Text: [Hello,
        DotNetVault! Accessing large structs by reference is efficient!].
Printing unaffected copy of protected resource: [MyMutableStruct] -- Time: [2020-02-13T13:34:31.9985334-05:00], Count:
        [0], Text: [Hello, DotNetVault!].
Going to change the value of the copy so count is 12 and text is `in lecto sunt tuo!'.  This will not affect protected
        resource.
Printing protected resource: [MyMutableStruct] -- Time: [2020-02-13T13:34:31.9985334-05:00], Count: [1], Text: [Hello,
        DotNetVault! Accessing large structs by reference is efficient!].
Printing independently mutated copy of protected resource: [MyMutableStruct] -- Time: [2020-02-13T13:34:31.9985334-
        05:00], Count: [12], Text: [in lecto sunt tuo!].
We can, of course, work on the copy and then overwrite the protected resource with it.  For large structs, it is probably
        best to just work through the .Value property.
now protected resource and copy will once again be the same...
Printing protected resource: [MyMutableStruct] -- Time: [2020-02-13T13:34:31.9985334-05:00], Count: [12], Text: [in lecto
        sunt tuo!].
Printing copy of protected resource: [MyMutableStruct] -- Time: [2020-02-13T13:34:31.9985334-05:00], Count: [12], Text:
        [in lecto sunt tuo!].
LargeMutableStructsAreNowEfficientProtectedResourcesWithReturnByReference functions properly.
```

*Figure 12* -- Output from Figure 11

### *v.* Locked Resource Objects of Mutable Resource Vaults (*LockedVaultMutableResource<MutableResourceVault<T>, T>* and *LockedMonVaultMutableResource<MutableResourceMonitorVault<T>, T>*)

These resources are associated with mutable resource vaults[56]. Unlike the BasicVault's *LockedResource*, this *LockedResource* comes with many restrictions, enforced by the static analyzer, to ensure that no mutable state from outside mingles with the protected resource and to ensure that no mutable state from the protected resource leaks outside of the vault. To facilitate this, all access to the resource is mediated through delegates annotated with attributes that are meaningful to the static analyzer. *TVault* denotes the owning vault's type and *T* denotes the protected resource's type.

[56] v. § 4.c.iii, *supra.*

1. Mutable Locked Resource Object Delegates

The delegate declarations from the project are laid out:

```
/// <summary>
/// A delegate used to perform a query on the current status of an object locked by a
/// <see cref="LockedVaultMutableResource{TVault,TResource}"/> object or by a custom version
/// thereof <seealso cref="StringBuilderVault"/>.
/// </summary>
/// <typeparam name="TResource">The type of protected resource</typeparam>
/// <typeparam name="TResult">The return type of the query (must be vault-safe)</typeparam>
/// <param name="res">the protected resource </param>
/// <returns>The result</returns>
/// <remarks>See <see cref="NoNonVsCaptureAttribute"/> and the limitations it imposes on the semantics of
/// delegates so-annotated.</remarks>
[NoNonVsCapture]
public delegate TResult VaultQuery<TResource, [VaultSafeTypeParam] TResult>(in TResource res);
/// <summary>
/// A delegate used to perform a query on the current status of an object locked by a
/// <see cref="LockedVaultMutableResource{TVault,TResource}"/> object or by a custom version
/// thereof <seealso cref="StringBuilderVault"/>.
/// </summary>
/// <typeparam name="TResource">The type of protected resource</typeparam>
/// <typeparam name="TResult">The return type of the query (must be vault-safe) </typeparam>
/// <typeparam name="TAncillary">An ancillary parameter, which must be a vault-safe value passed by
/// constant reference, used in the delegate.</typeparam>
/// <param name="res">the protected resource </param>
/// <param name="ancillary">an ancillary vault-safe value passed by readonly-reference</param>
/// <returns>The result</returns>
/// <remarks>See <see cref="NoNonVsCaptureAttribute"/> and the limitations it imposes on the semantics of
/// delegates so-annotated.</remarks>
[NoNonVsCapture]
public delegate TResult VaultQuery<TResource, [VaultSafeTypeParam] TAncillary, [VaultSafeTypeParam]
    TResult>(in TResource res, in TAncillary ancillary);
/// <summary>
/// Execute a potentially mutating action on the vault
/// </summary>
/// <typeparam name="TResource">the type of protected resource you wish to mutate</typeparam>
/// <param name="res">the protected resource on which you wish to perform a mutation.</param>
[NoNonVsCapture]
public delegate void VaultAction<TResource>(ref TResource res);
/// <summary>
/// Execute a potentially mutating action on the vault
/// </summary>
/// <typeparam name="TResource">the type of protected resource you wish to mutate</typeparam>
/// <typeparam name="TAncillary">an ancillary type used in the delegate.  must be vault safe, must be passed
/// by readonly-reference.</typeparam>
/// <param name="res">the protected resource on which you wish to perform a mutation.</param>
/// <param name="ancillary">the ancillary object</param>
/// <remarks>See <see cref="NoNonVsCaptureAttribute"/> and the limitations it imposes on the semantics of
/// delegates so-annotated.</remarks>
[NoNonVsCapture]
public delegate void VaultAction<TResource, [VaultSafeTypeParam] TAncillary>(ref TResource res,
    in TAncillary ancillary);
```

*Figure 13* -- Special Delegates Used By *LockedVaultMutableResource* Objects to Prevent Leakage and Mingling of State

cont'd:

```
/// <summary>
/// Execute a mixed query: a value is desired and returned, but mutation may also happen to protected
/// resource duringthe operation
/// </summary>
/// <typeparam name="TResource">The protected resource type</typeparam>
/// <typeparam name="TAncillary">an ancillary type to be used in the delegate.  must be vault-safe,
///must be passed by
/// readonly-reference.</typeparam>
/// <typeparam name="TResult">The result type</typeparam>
/// <param name="res">the protected resource</param>
/// <param name="a">the ancillary object</param>
/// <returns>the result</returns>
/// <remarks>See <see cref="NoNonVsCaptureAttribute"/> and the limitations it imposes on the semantics
/// of delegates so-annotated.</remarks>
[NoNonVsCapture]
public delegate TResult VaultMixedOperation<TResource, [VaultSafeTypeParam] TAncillary,
    [VaultSafeTypeParam] TResult>(ref TResource res, in TAncillary a);

/// <summary>
/// Execute a mixed query: a value is desired and returned, but mutation may also happen to protected
/// resource during the operation
/// </summary>
/// <typeparam name="TResource">The protected resource type</typeparam>
/// <typeparam name="TResult">The result type</typeparam>
/// <param name="res">the protected resource</param>
/// <returns>the result</returns>
/// <remarks>See <see cref="NoNonVsCaptureAttribute"/> and the limitations it imposes on the semantics
/// of delegates so-annotated.</remarks>
[NoNonVsCapture]
public delegate TResult VaultMixedOperation<TResource, [VaultSafeTypeParam] TResult>(ref TResource
    res);
```

*Figure 14* -- (cont'd) Special Delegates Used By *LockedVaultMutableResource* Objects to Prevent Leakage and Mingling of State

The static analyzer enforces the following rules on all these delegates:

- Their return value, if any, must be vault safe.
- If an ancillary parameter is used, it must be vault safe
- Anything captured in the closure assigned to the delegate must be vault-safe
- Other than the protected resource, anything that is read-from or written-to must be vault safe
- No reference to the protected resource or anything not vault safe can be assigned to anything outside the delegate's body.

*VaultQueries* are used to change no state in the mutable resource but to return information regarding the current state of the protected resource. The protected resource is passed to the delegate by constant reference. *VaultActions* return no values but are used to change the state of the protected resource. One ancillary vault safe parameter is available. Only vault-safe captured values may be used. The protected resource is passed to the delegate by non-constant reference. *VaultMixedOperations* return values but also may change the state. The return value must be vault-safe. Any ancillary parameter used as well as anything captured in the closure must be vault-safe.

2. *Public Methods*

The LockedVaultMutableResource exposes the following methods (with overloads allowing for an optional vault-safe ancillary parameter):

- *ExecuteQuery*– accepts a *VaultQuery* delegate, an optional vault-safe ancillary parameter and returns a vault-safe value based on the logic of the delegate. The protected resource is passed to the delegate by constant reference.
- *ExecuteAction*– accepts a *VaultAction* delegate, optional vault-safe ancillary parameter and changes the state of the protected resource as specified by the delegate. The protected resource is passed to the delegate by (non-constant) reference.
- *ExecuteMixedOperation*– accepts a *VaultMixedOperation* delegate, an optional vault-safe ancillary parameter, returns a vault-safe return value as specified by the logic of the delegate and may change the state of the protected resource. The protected resource is passed to the delegate by (non-constant) reference.
- *Dispose*– returns the resource to the vault so it can be used by other threads. **This method cannot be called manually; it can only be called implicitly by the *using* syntax**.[57]
- *ErrorCaseReleaseOrCustomWrapperDispose*– a method used in narrow, highly specialized use cases (that require the calling method to have been annotated with the *EarlyReleaseJustificationAttribute*[58] on pain of compilation error) that releases the protected resource back to the Vault. Unless you understand the use cases[59], **do not use it**.

[57] v. §§ 5.f, 6.f, *infra.*
[58] v. § 6.h, *infra*.
[59] v. §§ 5.g, *infra*; v. Figure 26 and Figure 27, *infra*.

The best way to demonstrate the operation is to show examples of code with the output. For each example, assume that the protected resource is a *StringBuilder* that begins with the contents **"Hello, world!"**.

```
public static void DemonstrateQueries()
{
    const string methodName = nameof(DemonstrateQueries);
    Console.WriteLine();
    Console.WriteLine($"Performing {methodName}...");
    MutableResourceVault<StringBuilder> vault = CreateExampleVault();
    using var lck = vault.SpinLock();
    string contents = lck.ExecuteQuery((in StringBuilder res) => res.ToString());
    Console.WriteLine($"Contents: {contents}.");
    int ancillaryValue = 7;
    int lengthPlusAncillaryValue =
        lck.ExecuteQuery((in StringBuilder sb, in int anc) => sb.Length + anc,
            ancillaryValue);
    Console.WriteLine("Length of contents (content length: " +
                      $"[{lck.ExecuteQuery( (in StringBuilder sb) => sb.Length)}]) " +
                      $"plus [{nameof(ancillaryValue)}] of [{ancillaryValue}]: " +
                      $"{lengthPlusAncillaryValue.ToString()}");
    int idx = 1;
    char offSet = (char) 32;
    char valOfCharSpecifiedIndexMadeUppercase =
        lck.ExecuteQuery(
            (in StringBuilder sb, in char offset) =>
                (char) (sb[idx] - offSet), offSet);
    Console.WriteLine($"Char at idx [{idx.ToString()}] " +
                      $"(current val: [{contents[1]}]) made upper " +
                      $"case: [{valOfCharSpecifiedIndexMadeUppercase}].");
    Console.WriteLine();
}
```

*Figure 15* -- VaultQuery Demonstration

```
Performing DemonstrateQueries...

Contents: Hello, world!.

Length of contents (content length: [13]) plus [ancillaryValue] of [7]: 20

Char at idx [1] (current val: [e]) made upper case: [E].
```

*Figure 16* -- VaultQuery Demo Output

```
public static void DemonstrateActions()
{
    const string methodName = nameof(DemonstrateActions);
    Console.WriteLine();
    Console.WriteLine($"Performing {methodName}...");
    MutableResourceVault<StringBuilder> vault = CreateExampleVault();
    using var lck = vault.SpinLock();
    lck.ExecuteAction((ref StringBuilder res) =>
    {
        for (int i = 0; i < res.Length; ++i)
        {
            char current = res[i];
            if (char.IsLetter(current))
            {
                res[i] = char.IsLower(current) ? char.ToUpper(current) : char.ToLower(current);
            }
        }
     });
     string contents = lck.ExecuteQuery((in StringBuilder sb)
        => sb.ToString());
     Console.WriteLine("Reversed Upper/Lower res: " +
                       $"{contents}");
     //now let's make every char at idx divisible by three change to q
     int divisibleBy = 3;
     lck.ExecuteAction((ref StringBuilder res, in int d) =>
     {
         for (int i = 0; i < res.Length; ++i)
         {
             if (i % d == 0)
             {
                 res[i] = 'q';
             }
         }
      }, divisibleBy);
      contents = lck.ExecuteQuery((in StringBuilder sb)
            => sb.ToString());
      Console.WriteLine($"Made chars at idx divisble by 3 q: " +
                        $"[{contents}]");
      Console.WriteLine();
}
```

*Figure 17* -- VaultAction Demonstration

```
Performing DemonstrateActions...

Reversed Upper/Lower res: hELLO, WORLD!

Made chars at idx divisble by 3 q: [qELqO,qWOqLDq]
```

*Figure 18* -- VaultAction Demo Output

```
[VaultSafe]
public struct IndexAndVal
{
    public readonly string NewValue;
    public readonly int Idx;

    public IndexAndVal(string newVal, int idx)
    {
        NewValue = newVal;
        Idx = idx;
    }
}

public static void DemonstrateMixedOperations()
{
       const string methodName = nameof(DemonstrateActions);
       Console.WriteLine();
       Console.WriteLine($"Performing {methodName}...");
       MutableResourceVault<StringBuilder> vault = CreateExampleVault();
       using var lck = vault.SpinLock();

       //Find the index of the first char equal to 'o' and insert
       //" it's magic oooh oooh! " after it.
       //return the index and the new contents.
       //the number of insertions that were made.
       string insertMe = " it's magic oooh oooh! ";
       char queryLetter = 'o';
       IndexAndVal res = lck.ExecuteMixedOperation(
       (ref StringBuilder sb, in char ql) =>
       {
           string newVal;
           int idx = -1;
           for (int i = 0; i < sb.Length; ++i)
           {
               if (sb[i] == ql)
               {
                   idx = i;
                   break;
               }
           }

           if (idx != -1)
           {
               sb.Insert(idx + 1, insertMe);
           }

           return new IndexAndVal(sb.ToString(), idx);
       }, queryLetter);

       Console.WriteLine($"New value: [{res.NewValue}], " +
                         $"Index: [{res.Idx}].");
       Console.WriteLine();
}
```

*Figure 19* – VaultMixedOperation Demonstration

```
Performing DemonstrateMixedOperations...

New value: [Hello it's magic oooh oooh! , world!], Index: [4].
```

*Figure 20* -- VaultMixedOperation Demo Output

As can be seen, interacting with protected mutable resources requires more care and requires understanding of expression and statement lambdas. As mentioned above, facilities exist to create customized, more convenient *Locked Resources* and code generation utilities are planned for future released.[60] A shorter path to a more convenient syntax is also available: passing the locked resource to extension methods (by reference). The following example shows how extension methods can be used to simplify frequently used syntax:



[60] v. § 4.c.iv, *supra.*

```
public static void DemonstrateUseOfExtensionMethodsToSimplify()
{
    const string methodName = nameof(DemonstrateUseOfExtensionMethodsToSimplify);
    Console.WriteLine();
    Console.WriteLine($"Performing {methodName}...");
    MutableResourceVault<StringBuilder> vault = CreateExampleVault();
    using var lck = vault.SpinLock();
    Console.WriteLine("Contents: {0}", lck.GetContents());
    Console.WriteLine("First char: {0}", lck.GetCharAt(0));
    //make second char uppercase E
    lck.SetCharAt(1, 'E');
    Console.WriteLine("Changed to uppercase 'E': {0}", lck.GetContents());
    Console.WriteLine();
}

internal static class LockedSbResExtensions
{

    public static char GetCharAt(
        this in LockedVaultMutableResource<MutableResourceVault<StringBuilder>,
            StringBuilder> lck,  int idx) =>
        lck.ExecuteQuery((in StringBuilder sb, in int i) => sb[i], idx);

    public static void SetCharAt(
        this in LockedVaultMutableResource<MutableResourceVault<StringBuilder>,
            StringBuilder> lck, int idx, char newC)
        => lck.ExecuteAction((ref StringBuilder sb, in int i) => sb[i] = newC, idx);

    public static string GetContents(
        this in LockedVaultMutableResource<MutableResourceVault<StringBuilder>,
        StringBuilder> lck) => lck.ExecuteQuery((in StringBuilder sb) =>
              sb.ToString());
}
```

*Figure 21* – Demonstration of Extension Methods to Simplify Usage

```
Performing DemonstrateUseOfExtensionMethodsToSimplify...

Contents: Hello, world!

First char: H

Changed to uppercase 'E': HEllo, world!
```

*Figure 22* -- Output of Extension Method Demo

# 5. Static Analyzer Rules

The static analyzer facilitates isolation of protected resources and prevents inadvertent failure to Dispose a *LockedResource*.

a. DotNetVault_UsingMandatory
   DotNetVault_UsingMandatory_DeclaredInline

These rules require that the caller to any method that annotates its return type with the *UsingMandatoryAttribute*

1- Protect that return value with a using statement or declaration immediately. Failure to do so will cause a compilation error and
2- Declare the assignment target inline in the using statement or declaration.

Failure to adhere to either of the foregoing requirements causes a compilation error. *LockedResources* represent a lock that has been obtained on a protected resource that more than one thread wishes to access. Allowing it to be held open for longer than its lexical scope can starve out other threads and potentially result in the inability of any other thread to ever be able to access the locked resource again. If no timeout is used, it would result in deadlock. Use of the using statement is simple syntactically and guarantees that the lock will be disposed (and the resource returned to the vault) no later than the end of the scope in which it is received from the method returning the value. Requiring inline declaration of the assignment target prevents use-after-dispose (which would break thread-safety) and prevents reassignment (which could potentially break thread-safety).

b. DotNetVault_VaultSafe

This rule enforces that any type annotated with the vault-safe attribute[61], without its *OnFaith* property set to true, must strictly comply with all requirements imposed on vault-safe types. If you use the *OnFaith* property on a type it will be deemed vault-safe without analysis. This allows types that partially employ their own thread-safety mechanisms to be considered vault-safe by the analyzer. You must be careful because any changes to the type that break the type's built-in thread-safety or otherwise introduce constructs that are not thread-safe will not cause any compiler errors or warnings despite their affecting the vault-safety of the type.

[61] v. § 4.6.a, *infra*.

c. DotNetVault_VsDelegateCapture

This rule ensures that none of the LockedVaultMutableResource delegates[62] read or write to any captured or static variable that is not vault-safe except for the protected resource itself. This prevents any externally accessible mutable state from leaking outside or creeping into it. Within these delegates, it is an error to read from or write to anything save vault-safe data and the protected resource itself. It also prevents the use of non-vault-safe objects from being the "*this*" parameter of extension methods *regardless of whether the extension method invocation or normal static method invocation is used*.[63]

d. DotNetVault_VsTypeParams
DotNetVault_VsTypeParams_MethodInvoke
DotNetVault_VsTypeParams_ObjectCreate
DotNetVault_VsTypeParams_DelegateCreate

These rules enforce that a whenever a type argument is substituted for a type parameter annotated with the *VsTypeParamsAttribute* that type is vault safe. Essentially, it introduces a new type-constraint: that of vault-safety. The first rule enforces this in type declarations. The second whenever type arguments are provided to a generic method, the third at the instantiation of objects. The final rule applies when generic delegates are created. These rules are primarily applicable in the context of *LockedVaultMutableResource* delegates.[64]

e. DotNetVault_NotVsProtectable

Frequently a type's API will expect to receive collections of objects rather than individual objects, the most general of which is *IEnumerable<T>*. Neither *IEnumerable<T>* nor the most used collections in .NET's base class library are vault-safe. While, ideally, Immutable Collections (with vault-safe type arguments) would be used for updating protected resources, this could produce inconvenience and perhaps unacceptable performance on occasion.

[62] v. § 4.d.iii.1, *supra*.
[63] v. § 7.a, *infra*.
[64] *Id*.

To mitigate this problem, DotNetVault provides convenience types designed to wrap generic collections whose types are vault-safe with minimal performance overhead:

| vault-safe Convenience Wrappers | | |
|---|---|---|
| Declared Type | Use Case | Description |
| public struct ArrayEnumeratorWrapper<[VaultSafeTypeParam] T> : IEnumerator<T> | Takes an array of a vault-safe Type and provides a struct Enumerator to iterate it. Access to the underlying mutable collection is prevented. | Functions like any enumerator. As a struct enumerator, no unnecessary indirection or GC Pressure is introduced. |
| public struct KvpEnumerator<[VaultSafeTypeParam] TKey, [VaultSafeTypeParam] TValue> : IEnumerator<KeyValuePair<TKey, TValue>> | Takes a collection of KeyValuePair (where both TKey and TValue are vault-safe) and provides a struct enumerator to iterate it. Access to underlying mutable collection is prevented. | Functions like any enumerator. As a struct enumerator, no unnecessary indirection or GC Pressure is introduced. |
| public struct StandardEnumerator<[VaultSafeTypeParam] T> : IEnumerator<T> | Take any IEnumerator<T> of a vault-safe type T and wrap it in a struct enumerator that prevents access to any underlying mutable collection | Functions like any enumerator. As a struct enumerator, no *additional* indirection or GC Pressure is introduced. |
| public struct StructEnumeratorWrapper<TWrappedEnumerator, [VaultSafeTypeParam] TItem> : IEnumerator<TItem> where TWrappedEnumerator : struct, IEnumerator<TItem> | Take any existing struct enumerator of a vault-safe type T and return an enumerator that will prevent access to the underling mutable type. | Functions like any enumerator. As a struct enumerator, no *additional* indirection or GC Pressure is introduced. |
| public sealed class VsArrayWrapper<[VaultSafeTypeParam] T> : IVsArrayWrapper<T> | Take any existing array of vault-safe type T, and wrap it in a class that allows readonly access to the array but prevents changes | Functions like an *IReadOnlyList*<T>. Some overhead and GC pressure introduced, but minimal compared to overhead introduced by copying a large collection. |
| public sealed class VsEnumerableWrapper<[VaultSafeTypeParam] T> : IVsEnumerableWrapper<T> | Take any existing collection of vault-safe values (IEnumerable<T>) and wrap it in a collection that prevents access to any underlying mutable collection | Functions like any *IEnumerable*<T>. Some overhead and GC pressure introduced, but minimal compared to overhead introduced by copying a large collection. |
| public sealed class VsListWrapper<[VaultSafeTypeParam] T> : IVsListWrapper<T> | Take any existing collection of vault safe values of type List<T> and wrap it in a read-only collection that prevents access to any underlying mutable collection | Functions like an IReadOnlyList<T>. Some overhead and GC pressure introduced, but minimal compared to overhead introduced by copying a large collection. |

*Figure 23* -- vault-safe Convenience Wrappers

These types are all annotated with the *VaultSafeAttribute* with its *OnFaith* property set to true but are also annotated with the *NotVsProtectableAttribute*[65]. The purpose of the wrapper is to permit vault-safe access to values of these types in the without fear of permitting access to the underlying mutable collection therein. *They are **not**, however,* thread-safe in any way. If you use them, you must be sure that no other threads have write access to these objects while they are being processed by the delegate. Because these objects do not provide real thread-safety or vault-safety (but instead a temporary

[65] v. § 6.e, *infra.*

façade of such safety), the *NotVsProtectableAttribute* forbids them from being a part of any protected resource in any vault. Their sole purpose is conveniently to prevent mingling or leaking mutable state with the protected resource.

This rule detects and forbids use of these objects as a protected resource in a vault or being a part, in any way, of such a protected resource. If you wish to protect a collection of vault-safe types, either use the BasicVault to protect an immutable collection thereof or use the *MutableResourceVault* and write extension methods or a customized vault permitting updates via immutable collections or these convenience wrappers.

The following shows the intended use-case for such wrappers:

```
public static void ShowWrapperUsage()
{
    Console.WriteLine("Begin showing wrapper usage.");
    MutableResourceVault<List<int>> vault =
        MutableResourceVault<List<int>>.CreateMutableResourceVault(() =>
                new List<int> { 1, 2, 3, 4 },
            TimeSpan.FromMilliseconds(250));
    ImmutableArray<int> finalContents;
    {
        using var lck =
            vault.SpinLock();
        List<int> addUs = new List<int> {5, 6, 7, 8};
        //ERROR DotNetVault_VsDelegateCapture cannot capute non-vault
        //safe param addUs, of type List, not vault-safe
        //lck.ExecuteAction((ref List<int> res) => res.AddRange(addUs));

        //Ok reference is to thin readonly wrapper around list of vault-safe type.
        //state cannot be commingled in the delegate.
        VsListWrapper<int> wrapper = VsListWrapper<int>.FromList(addUs);
        lck.ExecuteAction((ref List<int> res) => res.AddRange(wrapper));
        finalContents = lck.ExecuteQuery((in List<int> res) =>
            res.ToImmutableArray());
    }
    Console.WriteLine("Printing final contents: ");
    StringBuilder sb = new StringBuilder("{");
    foreach (var i in finalContents)
    {
        sb.Append($"{i}, ");
    }

    if (sb[^1] == ' ' && sb[^2] == ',')
    {
        sb.Remove(sb.Length - 2, 2);
    }

    sb.Append("}");
    Console.WriteLine(sb.ToString());

    Console.WriteLine("Done showing wrapper usage.");
    Console.WriteLine();
}
```

*Figure 24* – Usage of Vs Convenience Wrappers

```
Begin showing wrapper usage.
Printing final contents:
{1, 2, 3, 4, 5, 6, 7, 8}
Done showing wrapper usage.
```

*Figure 25* -- Usage Wrapper Demo Output

f. DotNetVault_NotDirectlyInvocable

The *Dispose* method of *LockedResources* is not intended to be called directly, but rather to be called indirectly because protected by a using construct. If called directly before the object's lifetime ends at the end of the using scope, unsynchronized access to the protected resource becomes possible. For this reason, these *Dispose* methods are annotated with the *NoDirectInvokeAttribute*. This rule causes a compilation error if an attempt is made to call the Dispose manually.[66]

g. DotNetVault_UnjustifiedEarlyDispose

There are two circumstances where it becomes necessary to manually free a *LockedResource* despite the DotNetVault_NotDirectlyInvocable rule above. Because these are relatively unusual situations that apply to a *LockedVaultMutableResource<TVault, TResource>* and generally calling Dispose manually is highly undesirable, a separate method for manual disposal was added to this object called *ErrorCaseReleaseOrCustomWrapperDispose*. This method is annotated with the *EarlyReleaseAttribute* which requires any method that calls *ErrorCaseReleaseOrCustomWrapperDispose* to be itself annotated with the *EarlyReleaseJustificationAttribute*[67] to prevent accidental calling outside the narrow circumstances that call for its use. The two acceptable circumstances are described by the enum values *EarlyReleaseReason.DisposingOnError* and *EarlyReleaseReason.CustomWrapperDispose*.

*i. EarlyReleaseReason.DisposingOnError*

This reason denotes that a method that would ordinarily return a *LockedResource* protected by the *UsingMandatoryAttribute* cannot return it because of an exception. If the *LockedResource* has already been created before the exception occurs, it must be released before the exception is rethrown or the protected resource will be rendered forever inaccessible. The *EarlyReleaseJustificationAttribute* should annotate the public *Lock* and *SpinLock* methods of all Custom Vaults and the *UsingMandatoryAttribute* should annotate their return value. *T*he *StringBuilderVault*'s[68] *Lock* method is shown as a paradigm:

[66] v. § 7.a, *infra.*
[67] v. §§ 6.g-h, *infra.*
[68] v. § 4.c.iv, *supra.*

```
[return: UsingMandatory]
[EarlyReleaseJustification(EarlyReleaseReason.DisposingOnError)]
public LockedStringBuilder Lock(TimeSpan timeout)
{
    LockedVaultMutableResource<MutableResourceVault<StringBuilder>, StringBuilder> temp = default;
    try
    {
        temp =
            _impl.GetLockedResourceBase(timeout, CancellationToken.None, false);
        return LockedStringBuilder.CreateLockedResource(temp);
    }
    catch (ArgumentNullException e)
    {
        temp.ErrorCaseReleaseOrCustomWrapperDispose();
        DebugLog.Log(e);
        throw;
    }
    catch (ArgumentException inner)
    {
        temp.ErrorCaseReleaseOrCustomWrapperDispose();
        DebugLog.Log(inner);
        throw new InvalidOperationException("The vault is disposed or currently being disposed.");
    }
    catch (Exception e)
    {
        temp.ErrorCaseReleaseOrCustomWrapperDispose();
        DebugLog.Log(e);
        throw;
    }
}
```

*Figure 26* – If the resource is not manually released before exceptions rethrown, it will be forever inaccessible.

### ii. *EarlyReleaseReason.CustomWrapperDispose*

Custom locked resource objects, such as the *LockedStringBuilder*,[69] simply store a wrapped LockedVaultMutableResource object.[70] When such custom locked resources are disposed (implicitly as a result of the scope of their *using* construct expiring), they simply release the object they wrap. Thus, the *LockedStringBuilder*'s *Dispose* method and the *Dispose* methods of all such custom locked resource objects should be annotated both with the *NoDirectInvokeAttribute* and the *EarlyReleaseJustificationAttribute* as shown:

```
/// <summary>
/// Release the lock on the <see cref="StringBuilder"/>, returning it to the vault
/// for use on other threads
/// </summary>
[NoDirectInvoke]
[EarlyReleaseJustification(EarlyReleaseReason.CustomWrapperDispose)]
public void Dispose() => _resource.ErrorCaseReleaseOrCustomWrapperDispose();
```

*Figure 27* – Shows how to annotate the Dispose method of custom locked resource objects.

---

[69] v. § 4.c.iv, *supra.*

[70] In *LockedStringBuilder's* case, the wrapped object is a *LockedVaultMutableResource<MutableResourceVault<StringBuilder>, StringBuilder>*

h. DotNetVault_NoExplicitByRefAlias

Above, it is demonstrated that allowing a ref local to alias the value property of a locked resource can result in unsynchronized access to that resource. Thus, this rule prevents ref local aliasing of any property whose return value is annotated with the BasicVaultProtectedResourceAttribute. This attribute annotates the Value property of the LockedVaultObject<TVault, [VaultSafeTypeParam] T> to prevent such unsynchronized access.

# 6. Attributes

a. *VaultSafeAttribute*

The vault-safe attribute when placed on a class or struct indicates to the static analyzer that the type so-annotated is vault-safe. A default-constructed instance of this attribute, or one with the first constructor parameter evaluating to *false*, will have the static analyzer verify that the type is provably vault-safe.[71] If constructed with its first constructor parameter evaluating to *true*, the static analyzer shall assume, without further analysis, it to be vault-safe. Note that types that comply with the *unmanaged* type constraint are deemed vault-safe regardless of the presence or absence of this attribute.

There is also included herewith a whitelist file called "VaultSafeWhiteList.txt" with the names of types should be deemed vault-safe by the analyzer without analysis. You can find this file in "Users/[*YourUserName*]/AppData/Local/VaultAnalysisData". The following shows the by-default contents of this whitelist:

```
String; System.Uri;
```

*Figure 28* -- -- Contents of Whitelist.txt

There is a second whitelist file for conditionally vault-safe generic types called "condit_generic_whitelist.txt". This file is also found at "Users/[*YourUserName*]/AppData/Local/VaultAnalysisData" These types will be deemed vault-safe without analysis **if and only if** *all their generic type arguments are vault-safe*. This is included to allow the use of immutable collections (whose generic

[71] For the characteristics of vault-safe types see § 4.a, *supra*.

type arguments are all vault-safe) as vault-safe types. If you design any generic immutable collections, they will be deemed vault-safe whenever all their type arguments are vault-safe. The starting contents of the file are shown below:

```
System.Collections.Immutable.ImmutableHashSet`1;
System.Collections.Immutable.ImmutableList`1+Enumerator;
System.Collections.Immutable.ImmutableQueue`1+Enumerator;
System.Collections.Immutable.ImmutableArray`1+Enumerator;
System.Collections.Immutable.ImmutableHashSet`1+Enumerator;
System.Collections.Immutable.ImmutableSortedSet`1;
System.Collections.Immutable.ImmutableDictionary`2+Enumerator;
System.Collections.Immutable.ImmutableSortedDictionary`2+Enumerator;
System.Collections.Immutable.ImmutableList`1;
System.Collections.Immutable.ImmutableStack`1+Enumerator;
System.Collections.Generic.KeyValuePair`2;
System.Collections.Immutable.ImmutableQueue`1;
System.Collections.Immutable.ImmutableArray`1;
System.Collections.Immutable.ImmutableSortedSet`1+Enumerator;
System.Collections.Immutable.ImmutableStack`1;
System.Collections.Immutable.ImmutableDictionary`2;
System.Collections.Immutable.ImmutableSortedDictionary`2;
```

*Figure 29*-- Contents of condit_generic_whitelist.txt

b. *UsingMandatoryAttribute*

This attribute may be used to annotate the return value of a method. When so annotated, it becomes a compiler error for the receiving attribute to fail to both declare and assign the variable in the context of a *using* statement or declaration.[72] Pre-declaration of the variable will cause compilation failure.

c. *VaultSafeTypeParamAttribute*

This attribute can be applied to type parameters. It effectively serves as a type constraint, enforced by the static analyzer, that any type argument substituted for the parameter must be vault-safe. It is considered an error for such a type not to be vault-safe. This may attribute may annotate type parameters wherever they are declared: on generic types, generic methods and generic delegates.

[72] v. § 5.a, *supra*.

d. *NoNonVsCaptureAttribute*

This attribute annotates delegates causing the static analyzer to ensure that all variables, not explicitly passed to it as method arguments, accessed by the delegate are vault-safe. It is an error to capture or otherwise read to or write from any variable from within the delegate unless the variable is:

1- vault-safe or
2- passed as a type argument

The LockedVaultMutableResource delegates[73] are all annotated with this attribute. All type parameters passed, except the first, which is the protected resource, are annotated with the *VaultSafeTypeParamAttribute*. This combination of attributes prevents any non-vault-safe variables, other than the protected resource, from being read from or written to in the delegate.

e. *NotVsProtectableAttribute*

This attribute may be applied to a type. Any type – even a vault-safe type – so annotated cannot be used as a protected resource or part of a protected resource in any vault. If the type is otherwise vault-safe, it will be considered vault-safe only for purposes of the rules at § 5.b-d, *supra*. This attribute is used primarily to allow for convenience wrappers around mutable collections of vault-safe type to be used in LockedVaultMutableResource delegates.

f. *NoDirectInvokeAttribute*

When this attribute annotates a method, the method may not be called directly. Such objects are annotated with the UsingMandatoryAttribute meaning that they must be guarded by a using construct. The problem arose that if one called Dispose manually, one could then still access the protected resource while the variable remained in scope even though it was no longer protected, potentially resulting in unsynchronized access to the protected resource. This attribute solves the problem by prohibiting direct invocation of the method -- it may only be called indirectly via using. Every *LockedResource's* Dispose() method is annotated with this attribute. Any Customized *LockedResources* you create should annotate their Dispose method with this attribute to guarantee prevention of unsynchronized access to the resource.

[73] v. § 4.d.iii.1, *supra*.

g. *EarlyReleaseAttribute*

The *LockedVaultMutableResource<TVault, TResource>* has a method permitting direct programmatic release of the protected resource. This method is called *ErrorCaseReleaseOrCustomWrapperDispose*, which releases the protected resource without using the *Dispose* method, whose direct invocation is prohibited by the *NoDirectInvokeAttribute*. This attribute decorates this method and requires any method that calls the *ErrorCaseReleaseOrCustomWrapperDispose* to justify the reason for the manual disposal by being annotated with the *EarlyReleaseJustificationAttribute*.[74] The need for early disposal arises in two circumstances:[75]

- custom locked resource objects that wrap a *LockedVaultMutableResource<TVault, TResource>* have a method that simply call's the wrapped object's *ErrorCaseReleaseOrCustomWrapperDispose* method

- thrown exceptions in the *Lock* and *SpinLock* methods after the creation of the object need to cleanup the object they dispose before rethrowing the exception

h. *EarlyReleaseJustificationAttribute*

This attribute must decorate any method that contains any method call to a method annotated with the *EarlyReleaseAttribute*. This attribute's constructor requires a parameter of type EarlyReleaseReason, an enum with the following values: DisposingOnError and CustomWrapperDispose.[76]

i. *BasicVaultProtectedResourceAttribute*

This attribute annotates the *Value* property of the *LockedVaultObject<TVault, [VaultSafeTypeParam] T>*[77] locked resource object to prevent aliasing into a ref-local. Such aliasing must be avoided because it can permit unsynchronized access to the protected resource.[78]

[74] v. EarlyReleaseJustificationAttribute, *infra.*
[75] v. Figure 26 and Figure 27, *supra.*
[76] v. EarlyReleaseAttribute, *supra.*
[77] v. § 4.d.ii, *supra.*
[78] v. § 5.h, *supra*.

# 7. Known Flaws and Limitations

## a. *Table of Known Issues*

*Table 4*

| Known Flaws and Limitations | | | | | |
|---|---|---|---|---|---|
| Name | Number | Description | Workaround / Fix | Bug # | Status |
| Double dispose | 1 | Possible Data Race with double dispose | Direct call of Dispose has been forbidden.[79] | 61 | FIXED |
| Potentially Mutating Extension method | 2 | Extension methods treated differently based on invocation syntax used | Analysis now prohibits passing non-vault-safe objects to extension methods regardless of whether passed via static or extension method syntax.[80] | 62 | FIXED |
| Potentially Leaking Member Functions in Protected Resources | 3 | For types that are not vault-safe, the analyzer cannot detect if the types non-static methods leak resources. | Ensure that non-static member functions do not leak shared mutable state before calling them -- even in LockedVaultMutableResource delegates. There is no reasonable way to prevent this, though it should not be a problem in most cases. | N/A | N/A |
| Immutable reference type with setter in vault safe struct | 4 | This should not render the struct not vault-safe but it does | Fixed the analyzer to address this | 64 | FIXED |
| Local Function Capture Not Detected | 5 | Illegal reference to non-vault safe types inside local functions and anonymous methods not detected as illegal | Fixed the analyzer to address this | 76 | FIXED |

[79] v. §§ 5.f, 6.f, *supra.*
[80] v. § 5.c, supra.

*b. Example Code Showing Problems*

```
public static void DemonstrateDoubleDispose()
{
       using (var bv = new BasicVault<DateTime>(DateTime.Now,
              TimeSpan.FromMilliseconds(250)))
    {
        {
            using var lck = bv.Lock();
            Console.WriteLine(lck.Value.ToString("O"));
            lck.Value = lck.Value + TimeSpan.FromDays(25);
            //BUG 61 -- potential race condition / Known Flaw #1
            //POTENTIAL RACE CONDITION -- DO NOT DISPOSE MANUALLY IN THIS MANNER!
            // BUG 61 -- potential race condition / Known Flaw #1 FIXED
            // FOLLOWING LINE WILL NOT COMPILE
            lck.Dispose();
            Console.WriteLine(lck.Value.ToString("O"));
        }
    }
}
```

*Figure* 30 – Double Dispose (Known Flaw #1 -- FIXED)

```
public static void DemonstrateBadExtensionMethod()
{
    using (var mrv =
        MutableResourceVault<StringBuilder>.CreateMutableResourceVault(() => new
                StringBuilder(), TimeSpan.FromMilliseconds(250)))
    {
        int lengthOfTheStringBuilder;
        {
            using var lck = mrv.Lock();
            // BUG# 62 Known Flaw# 2 FIXED – following 2 lines will no longer compile:
            lengthOfTheStringBuilder = lck.ExecuteQuery((in StringBuilder sb) =>
                        sb.GetStringBuilderLength());
        }
        Console.WriteLine($"The length of the string builder is:
                {lengthOfTheStringBuilder.ToString()}.");
        //BUG# 62 Known Flaw# 2 BAD -- Potential Race Condition on Protected Resource
        //BUG BAD -- if this were multithreaded defeats vault causing race condition -- I
         //altered protected resource outside vault
        Flaw2StringBuilderExtensions.LastEvaluatedStringBuilder.AppendLine("Hi mom!");

        //Show the protected resource has been effected without accessing lock:
        {
            using var lck = mrv.Lock();
            string printMe = lck.ExecuteQuery((in StringBuilder sb) => sb.ToString());
            Console.WriteLine($"BAD -- look, we mutated the protected resource:
                        [{printMe}].");
        }
    }
}
public static class Flaw2StringBuilderExtensions
{
    public static StringBuilder LastEvaluatedStringBuilder { get; set; }

    //BAD -- LEAKS SHARED MUTABLE STATE
    public static int GetStringBuilderLength(this StringBuilder sb)
    {
        LastEvaluatedStringBuilder = sb ?? throw new ArgumentNullException(nameof(sb));
        return sb.Length;
    }
}
```

*Figure 31* – Bad Extension Method (Known Flaw #2 -- FIXED)

```
public sealed class BadArray : IEnumerable<DateTime>
{
    [CanBeNull] public static BadArray LastBadArrayCreatedOrUpdated { get; set; }
    public IEnumerator<DateTime> GetEnumerator() => _list.GetEnumerator();
    public int Count => _list.Count;
    public BadArray() => _list = new List<DateTime>();
    IEnumerator IEnumerable.GetEnumerator() => ((IEnumerable) _list).GetEnumerator();
    public void Add(DateTime item)
    {
        _list.Add(item);
        LastBadArrayCreatedOrUpdated = this;
    }
    public DateTime this[int index]
    {
        get => _list[index];
        set
        {
            _list[index] = value;
            LastBadArrayCreatedOrUpdated = this;
        }
    }
    [NotNull] private readonly List<DateTime> _list;
}
public static void DemonstrateBadArrayBecauseTypeItselfInherentlyLeaksOwnState()
{
    using (var mrv =
        MutableResourceVault<BadArray>.CreateMutableResourceVault(() => new BadArray(),
            TimeSpan.FromMilliseconds(250)))
    {
        {
            using var lck = mrv.Lock();
            lck.ExecuteAction((ref BadArray ba) =>
            {
                DateTime now = DateTime.Now;
                ba.Add(now);
                ba.Add(now + TimeSpan.FromDays(1));
                ba.Add(now + TimeSpan.FromDays(2));
            });
        }
        //N.B. FLAW 3 This type inherently and shamelessly leaks shared mutable state.  The following
        //would affect the protected resource without obtaining lock and cause potential race condition!
        Console.WriteLine(
            $"The last date time in the last updated bad array is:
                [{BadArray.LastBadArrayCreatedOrUpdated?.Last().ToString("O") ?? "EMPTY"}]");
        //now we'll go ahead and mutate it without a lock
        BadArray.LastBadArrayCreatedOrUpdated?.Add(DateTime.Now + TimeSpan.FromDays(4));
        //demonstrate that the protected resource has been mutated without lock:
        string printedLastDt;
        {
            using var lck = mrv.Lock();
            printedLastDt = lck.ExecuteQuery((in BadArray ba) => ba.Last().ToString("O"));
        }
        Console.WriteLine($"The last element has changed to: [{printedLastDt}].");
    }
}
```

*Figure 32* – Bad Type Inherently Leaks (Known Flaw #3)

```
[VaultSafe]
public struct Bug64Demo
{
    /// <summary>
    /// Should NOT need to comment out setter to compile (and currently do not)
    /// </summary>
    public DateTime Timestamp
    {
        get;
        set;
    }

    /// <summary>
    /// Should NOT need to comment out  setter to compile (but currently do)
    /// Bug 64 FIX: NO LONGER NEED TO COMMENT IT OUT! (Intellisense still seems to think I do,
    /// Bug 64 but it builds.  Bizarre.) ... perhaps submit bug to ms
    /// Bug 64 Intellisense Note: intellisense does not seem to catch up with the compiler
    /// bug 64 after installing fixed library/analyzer until restarting visual studio
    ///
    /// </summary>
    public string StatusText
    {
        get;
        set; //Should Before Bug 64 fix, setter
             //wrongly made this type not-vault safe
    }
}


[VaultSafe]
public sealed class Bug64DemoCounterpoint
{
    /// <summary>
    /// Should need to comment out set to compile
    /// (Currently works correctly)
    /// </summary>
    public DateTime TimeStamp
    {
        get;
        //set;
    }


    /// <summary>
    /// Should need to comment out set to compile
    /// (Currently works correctly)
    /// </summary>
    public string StatusText
    {
        //BUG 64 COMMENT -- if this were a STRUCT, the setter would be ok
        //Bug 64 COMMENT -- since this is a reference type, it must be 100% immutable
        //Bug 64 COMMENT -- all the way through its graph
        //Bug 64 COMMENT -- otherwise, a copy to this object that escaped from the vault
        //Bug 64 COMMENT -- could be used to change the value in the vault without synchronization
        get;
        //set;
    }

    public Bug64DemoCounterpoint(DateTime ts, string text)
    {
        TimeStamp = ts;
        StatusText = text;
    }
}
```

*Figure 33* – Shows Bug 64 Fix

**BEFORE** BUG 76 FIX:

```
static void Main()
{
    using (var vault = InitVault())
    {
        {
            using var lck = vault.Lock();
            //following line should not compile.  Following bug 76 fix, it will not.
            lck.ExecuteAction(AppendText, "Hi mom!");
        }
        StaticStringBuilder.AppendLine("I should not be able to do this.  After bug 76 fix, I won't be able to.");
        Console.WriteLine(StaticStringBuilder.ToString());
    }
    static void AppendText(ref StringBuilder sb, in string text)
    {
        //The following line illegally references a non-vault safe variable,
        //allowing the protected string builder to escape from the vault
        //and allowing unsynchronized mutation thereof. After bug 76 fix,
        //will not compile without removing the assignment.
        StaticStringBuilder = sb;
        sb.AppendLine(text);
    }
}
private static StringBuilder StaticStringBuilder = new StringBuilder();
```

OUTPUT:

```
Hi mom!
I should not be able to do this.  After bug 76, I won't be able to.

Press any key to continue . . .
```

**AFTER** BUG 76 FIX:

```
        using var lck :LockedVaultMutableResource<MutableResourceVault<StringBuilder>,…> = vault.Lock();
        //following line should not compile.  Following bug 76 fix, it will not.
        lck.ExecuteAction(AppendText, ancillaryValue:"Hi mom!");
```

COMPILATION ERROR SHOWN:

```
1>Build FAILED.
1>
1>"C:\Users\Christopher Susie\Source\Scratch\Bug76Demo\Bug76Demo\Bug76Demo\Bug76Demo.csproj"
(Build;BuiltProjectOutputGroup;BuiltProjectOutputGroupDependencies;DebugSymbolsProjectOutputGroup;De
bugSymbolsProjectOutputGroupDependencies;DocumentationProjectOutputGroup;DocumentationProjectOutputG
roupDependencies;SatelliteDllsProjectOutputGroup;SatelliteDllsProjectOutputGroupDependencies;SGenFil
esOutputGroup;SGenFilesOutputGroupDependencies target) (1) ->
1>(CoreCompile target) ->
1>  C:\Users\Christopher Susie\Source\Scratch\Bug76Demo\Bug76Demo\Bug76Demo\Program.cs(16,31,16,41):
error DotNetVault_VsDelegateCapture: The delegate is annotated with the NoNonVsCaptureAttribute
attribute but captures or references the following non-vault safe type- [Non - compliant symbol:
StringBuilder;       Explanation: The operation references the field StaticStringBuilder which is of
type StringBuilder, a type that is not vault-safe.
1>
1>    0 Warning(s)
1>    1 Error(s)
1>
1>Time Elapsed 00:00:01.05
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

*Figure 34* -- Demonstrates Bug 76 and its Fix

# 8. Licensing

## a. *Software License*

The DotNetVault software library, ancillary projects and their source-code are the intellectual property of CJM Screws, LLC, a Maryland Limited Liability Company. CJM Screws, LLC licenses the software and source code to you under the MIT License as follows:

```
MIT License

Copyright (c) 2019-2020 CJM Screws, LLC

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

## *b. Documentation License*

This document is the intellectual property of CJM Screws, LLC, a Maryland Limited Liability Company. CJM Screws, LLC hereby grants you the right to use, copy and distribute this document in unaltered form without limitation provided any persons to whom the documentation is distributed agree to be bound by the terms of this license including the following:

1- The documentation may not be altered in any material way and

2- THE DOCUMENTATION IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE DOCUMENTATION OR THE USE OR OTHER DEALINGS IN THE DOCUMENTATION.

*c. Author Contact Information*

The author of the documentation and software is Christopher P. Susie, a member of CJM Screws, LLC. He can be reached by email at cpsusie@hotmail.com or by mail at

CJM Screws, LLC
ATTN: Christopher P. Susie
3705 Ridgecroft Road
Baltimore, Maryland 21206
U.S.A.