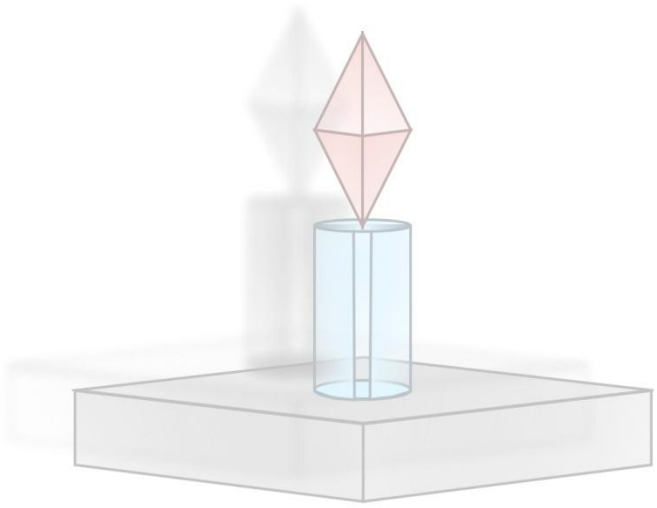
Attacking Interoperability



Version: 1.0
Black Hat USA 2009

Authors: Mark Dowd markdowd@au1.ibm.com
Ryan Smith ryan@hustlelabs.com
David Dewey dewey@us.ibm.com

Contents

Introduction

In recent years, there has been an increased drive for client applications to deliver tailored, dynamic content. Traditionally, this goal has been achieved by combining layout directives with scripting capabilities that can source data and programmatically modify the layout (such as HTML and JavaScript). This model has since evolved to allow a much finer grained collaboration between embedded objects, extended layout properties, and scripting engines (including those that are utilized within embedded objects, such as Adobe's ActionScript). This new wave of interoperability facilitates the creation of a seamless user experience that spans multiple technologies.

This paper intends to explore the security implications of software interoperability layers, focusing specifically on several prominent web browser technologies. We will expose vast and largely unexplored attack surfaces that are a direct result of permitting such interoperability, and discuss the unique types of vulnerabilities that are likely to be present in them. Furthermore, we will explore the impact that interoperability has on security features implemented in the host application. Specifically, we will demonstrate how these security features can often be undermined by pluggable components as a direct result of trust being extended to said components. Although the paper primarily focuses on several interoperability layers present within contemporary web browsers, much of the discussion about vulnerability classes and auditing strategies can be applied to a broad spectrum of software that performs some sort of inter-component data exchange. Some examples of such software would include other scripting languages and plugin architectures, RPC stacks, and virtual machines.

Organization of this Paper

This paper is divided into three parts. First, there will be a brief introductory tour of the attack surface that this paper seeks to address in Section 1. Specifically, the general browser architecture will be examined and components relevant to attacking interoperability will be highlighted. The second part of the paper, Section 2, will then provide a technology overview that provides background information for how interoperability works within two popular browsers: Microsoft Internet Explorer (IE) , and Mozilla Firefox. Finally, Section 3 will be dedicated to enumerating the classes of vulnerabilities that arise in the identified attack surfaces, and demonstrating practical strategies for uncovering these types of problems. A number of critical real-world vulnerabilities uncovered by the authors will be examined throughout this last section.

Section I: Attack Surface

Before delving into an in-depth discussion of targeted software layers, it is important to understand the attack surface from a conceptual level. Figure 1 represents a high-level architecture view of the contemporary web browser, with a breakdown of components that are relevant to this paper.

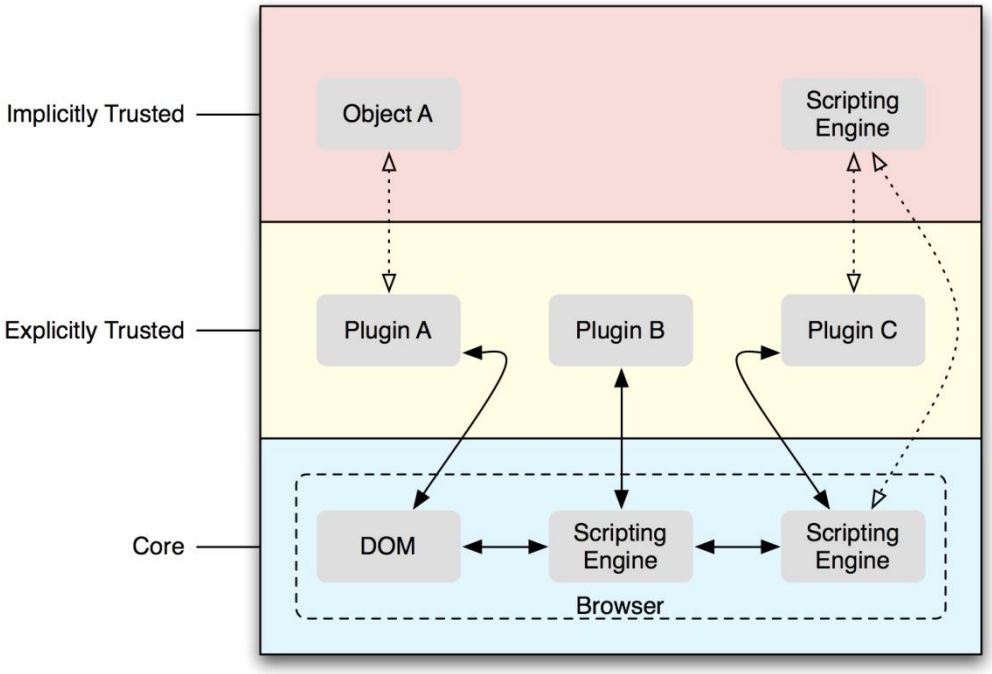


Figure 1: Architecture Model of Contemporary Web Browsers

Figure 1 is separated into three logical layers. The first layer, the browser core, contains several components that provide an environment for plugins to interact with the browser. Primarily, plugins are controlled through scripting, but they can also interact directly with the browser's Document Object Model (DOM) in certain situations.

The second layer represents the plugins themselves, which are essentially objects the browser loads to support additional functionality, primarily by handling unique document types. Plugins are explicitly granted or denied trust within the browser environment by the browser policy; however they sometimes run in a separate process context from the browser. For example, the Internet Explorer 8 (IE8) browser operates within a restrictive "Low Integrity" context when being run on Windows Vista or Windows 7, whereas several plugins run out of process in a less restrictive "Medium Integrity" context. This strategy allows the plugin to maintain full trust in the browser, but possess less trust in the context of the operating system.

Last, there is a third layer of implicitly trusted objects, which are those that trusted plugins may load to augment their own functionality. Since the browser explicitly extends trust to a plugin (plugin X), and the plugin extends trust to an arbitrary object (object Y), then we can say that there is a transitive trust relationship established between the browser and the arbitrary object loaded by the plugin (B -> X, X -> Y, therefore B -> Y). We will show examples in section two of this paper that the extension of such trust allows an attacker to utilize plugins and their trusted components to undermine the browser security model. Another noteworthy part of the third layer is that some plugins create their own scripting functionality, which in many cases can be used to interact with the scripting engines or DOM provided by the browser. Indeed, this situation is the case for a number of popular plugins, including Adobe Flash, Sun Java, and Microsoft Silverlight. In each case, trust is extended from the browser implicitly to allow an attacker to gain access to functionality that each scripting language provides. Furthermore, objects can be exported from those scripting languages back to scripting contexts within the browser. Due to trust transitivity, these objects might then be manipulated by not just browser scripting engines, but also DOM functions and other plugins as well, sometimes with quite unintended consequences.

Trust extension is not the only security cost of interoperability. From Diagram 1, we can see that for each additional component to interact with each other, a communications bridge must be established between the interoperating components. This is depicted in Diagram 1 by two-way arrows. These communication bridges are, in themselves, a rather large attack surface: it is the code responsible for marshalling data from one component to the next. The marshalling layer performs conversions implicitly between data structures native to the co-operating components. Since this layer operates somewhat silently, it is often overlooked when attempting to discover security flaws. In fact, there is currently a large volume of literature dedicated to evaluating plugin objects in browsers for security problems (with tangible results), but very little information about examining the interoperability layers. This lack of examination is one area that this paper will attempt to address.

Interoperability layers are a breeding ground for various unique vulnerability classes that have been largely unexplored previously. Due to the operations being performed, the marshalling infrastructure often lends itself to vulnerabilities related to type confusion (mis-use of data due to misinterpretation of its type) and object retention (spurious reference counting issues) issues that are seldom seen in other areas of an application. Although vulnerabilities like these have been occasionally uncovered in the past, we will show how the popular APIs in the targeted software are particularly vulnerable, and will provide strategies for uncovering these types of bugs in section two of this paper. It should be noted that, although the architectures mentioned in this paper are web browser-centric, these types of problems are systemic in any software that provides platforms for collaboration between components that have differing internal data representations.

Section II: Technology Overview

This section provides an overview of the relevant technologies that will be used as case studies to illustrate the concepts presented in the following section of this paper, "Attacking Interoperability". We include discussions of both Internet Explorer's ActiveX control architecture, as well as Mozilla's NPAPI plugin architecture (present in Firefox, Google Chrome, and several other non-browser applications). We will explore how objects are represented in the common scripting languages available, how they are marshaled and exported to plugin entry points, and how DOM interaction occurs. Lastly, we will provide an attack surface summary for both ActiveX and NPAPI that summarizes the roles each technology will play in the context of the stated attack surface.

Microsoft ActiveX

ActiveX is a technology derived from Microsoft's COM technology. It is utilized to create plugins that can be exposed to runtime engines (such as JavaScript and VBScript) to provide additional capabilities to the host application. Understanding the types of vulnerabilities that will be explored in section three of this paper requires an in-depth understanding of some of the COM / Automation architecture. As such, we will present an overview of the relevant technologies in this section. We will also explore the concept of "persistent objects", which are serialized COM objects that can be optionally embedded within web pages. It will be shown in section three how persistent COM objects can be used to not only target vulnerabilities in various COM marshalling components, but also undermine browser security features in certain scenarios.

Plugin Registration

ActiveX controls are a specialization of COM objects, and as such have an entry within the system registry describing the relevant instantiation information. Like any other COM object, each ActiveX object is identified by a globally unique Class ID (CLSID), and is located in the registry at HKEY_CLASSES_ROOT\CLSID\{<CLSID>}. Objects can also be installed on a per-user basis, using the HKEY_CURRENT_USER portion of the registry. Since COM objects are used so pervasively throughout the Windows OS, Internet Explorer (IE) needs a way of restricting which COM objects are allowed to be launched through the web browser. The semantics of the safety mechanisms have gradually become more granular over time, and will briefly be described here.

ActiveX Plugins: Safety Controls

IE has several mechanisms for determining whether an ActiveX object has permission to run. Safety permissions for controls are divided into two categories: initialization and scripting. Initialization safety refers to whether or not the control is allowed to be instantiated based on data from a persistent COM stream (discussed in depth shortly). Scripting safety refers to whether the control may be manipulated via scripting APIs exposed at runtime. A complete overview of ActiveX security controls is available from Microsoft at http://msdn.microsoft.com/en-us/library/bb250471(VS.85).aspx, which is where most of the information in this section that wasn't reverse engineered is derived from.

Registry Controls

The first and most well-known method to mark a control as Safe For Scripting (SFS) or Safe For Initialization (SFI) is to add specific subkeys below the entry for the control in the registry. Two values can be added under the "Implemented Categories" subkey to mark the control SFS and SFI respectively. These values are 7DD95801-9882-11CF-9FA9-00AA006C42C4 (CATID_SafeForScripting) and 7DD95802-9882-11CF-9FA9-00AA006C42C4 (CATID_SafeForInitialization) respectively. Figure 2 shows an example of a control using these categories.

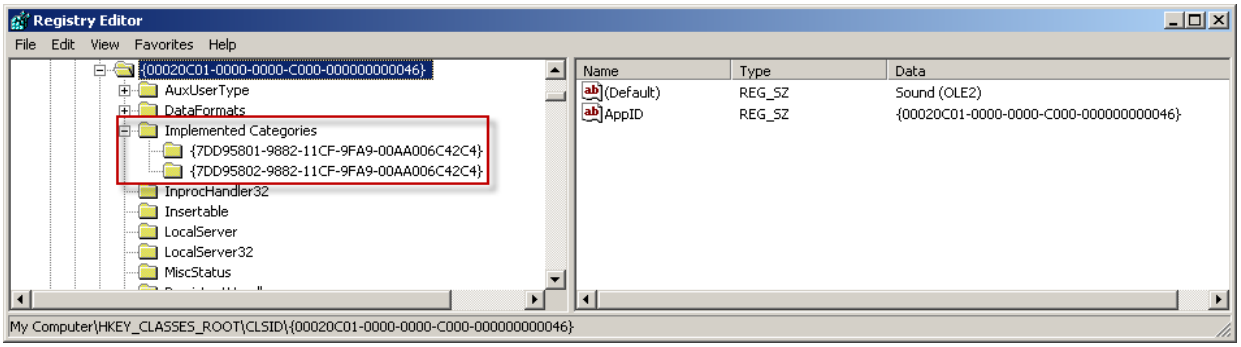


Figure 2: ActiveX control marked as "Safe for Initialization" (SFI) and "Safe For Scripting" (SFS)

Controls may programmatically register themselves for these categories using the StdComponentCategoriesMgr object. The ICatRegister interface contains a RegisterClassImplCategories() method, which can be used to manipulate the category registration information for any given COM object. Internally, the StdComponentCategoriesMgr updates the registry with the above information.

Internet Explorer utilizes the StdComponentCategoriesMgr object as well, but for enumeration rather than registration. The ICatInformation interface provides a function named IsClassOfCategories(), which IE can call to determine if a control is SFS or SFI. Again, this operation internally queries the above mentioned registry location to determine which controls the object implements.

Component category management is treated in depth at http://msdn.microsoft.com/en-us/library/ms692689(VS.85).aspx.

IObjectSafety Control

An alternative method exists to mark a control as SFS or SFI. An ActiveX control can provide support for either of these safety restrictions by implementing the IObjectSafety interface. In this scenario, the security capabilities for the control can be obtained by calling the IObjectSafety::GetInterfaceSafetyOptions() method, which has the following prototype.

```
HRESULT IObjectSafety::GetInterfaceSafetyOptions(
      REFIID riid,
      DWORD *pdwSupportedOptions,
      DWORD *pdwEnabledOptions
);
```

This function will be called by IE to determine the supported set of safety options. If the interface appears to support the security options, IE will then call the SetInterfaceSafetyOptions() method of the IObjectSafety interface with the options that it would like the object to enforce. SetInterfaceSafetyOptions has the following prototype.

```
HRESULT IObjectSafety::SetInterfaceSafetyOptions(
    REFIID riid,
    DWORD dwOptionSetMask,
    DWORD dwEnabledOptions
);
```

If SetInterfaceSafetyOptions() returns successfully, then the application can use the COM object knowing that the object intends to use the security options requested. The added value of this API over COM categories is that a control can offer more granular control over how it is used, since it is able to specify different security settings for different interfaces, based on which interface id was specified in the riid parameter for the method calls. Also, the IObjectSafety interface can execute native code to determine if the application that is creating the object can do so safely. A specific example of this type of functionality is the SiteLock template code provided by Microsoft. This template code allows the programmer to restrict ActiveX controls to a pre-determined list of URLs.

ActiveX Killbits

IE also implements an override to the standard safety features, allowing administrators to specifically ban the instantiation of selected controls within the browser. This is achieved by adding a subkey into the HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\ActiveX Compatibility registry location. The subkey added must have the CLSID of the control in question, and contain the DWORD value "Compatibility Flags", which has the "killbit" set (value 0x400). Figure 3 shows an example of a control with the killbit set.

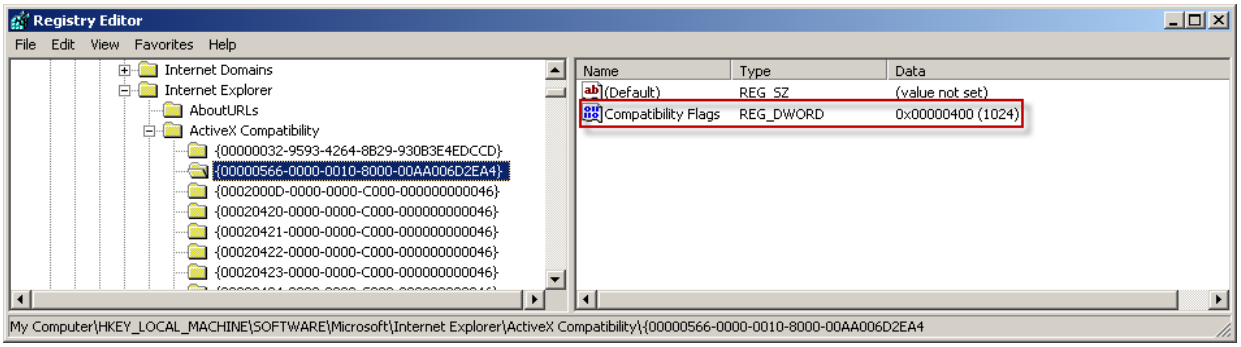


Figure 3: ActiveX Killbits in IE

When an application wishes to determine if the killbit is set, it will call the CompatFlagsFromClsid() function, which is exported from urlmon.dll. CompatFlagsFromClsid() has the following prototype:

```
HRESULT CompatFlagsFromClsid(
    CLSID *pclsid,
    LPDWORD pdwCompatFlags,
    LPDWORD pdwMiscStatusFlags
);
```

When the application calls this function, it will pass in the CLSID of the COM object it is interested in, and two DWORD pointers whose value will be equal to the compatibility and miscellaneous OLE flags for the object upon the successful return of the function. The application will then test to see if the 0x400 bit is set to determine if the control has the killbit set.

If the Killbit is set, then an entry may appear in the registry for an alternate class id. This alternate class id will be used in lieu of the original class id within Internet Explorer. Figure 4 shows a registry entry for a class id that uses an alternate class id. When dealing with the control in Figure 4, Internet Explorer will transparently translate requests for COM objects with a class id of {41B23C28-488E-4E5C-ACE2-BB0BBABE99E8} to the class id of {52A2AAAE-085D-4187-97EA-8C30DB990436}.

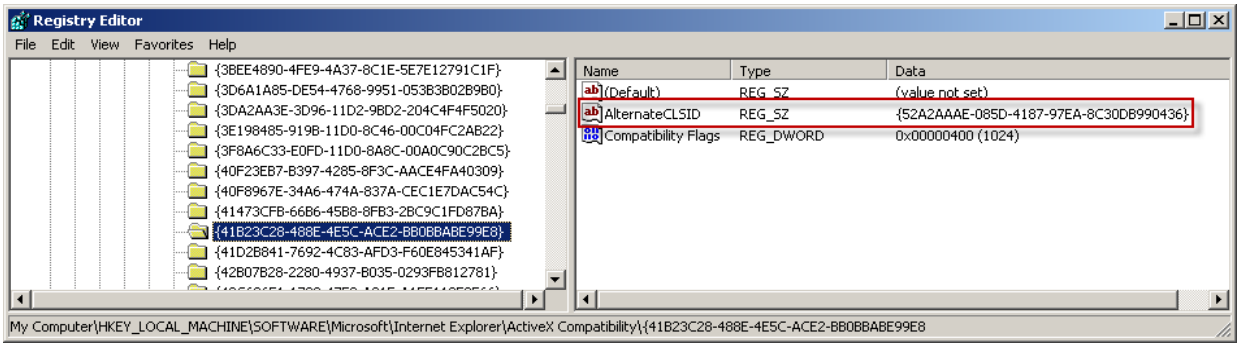


Figure 4: COM object with an alternate CLSID

Preapproved List / ActiveX Opt-In

Microsoft introduced a feature called ActiveX Opt-In with Internet Explorer 7. ActiveX Opt-In is designed to reduce the attack surface of the browser by prompting the user before a web page is allowed to instantiate an object that hasn't been loaded before in Internet Explorer, or wasn't installed by the user through Internet Explorer. Figure 5 shows the relevant area of the registry: HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Ext\PreApproved.

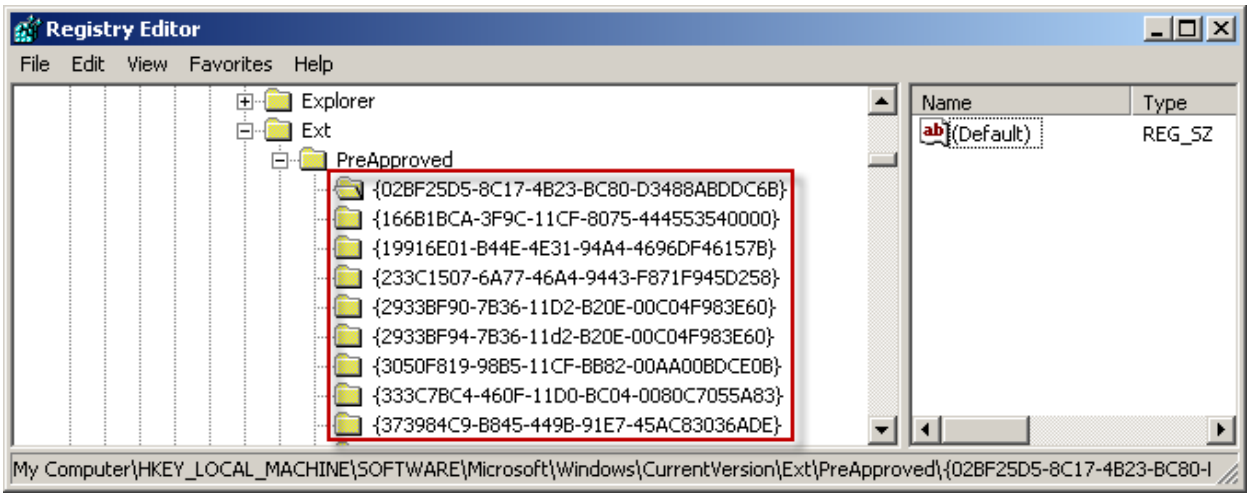


Figure 5: Excerpt of the preapproved list

In a base installation of Windows there are a number of controls already on the preapproved list. However, there are far more controls that are safe for scripting or initialization that do not appear on this list. This functionality makes it more desirable to find flaws in controls on this list, rather than flaws in other controls.

Per-User ActiveX Security

IE8 introduced a series of additional security capabilities related to secure browsing, including some refinements to ActiveX. Before these capabilities were added, control permissions that could be configured were configured on a per-machine basis. The new capabilities extend the per-machine killbit

to a per-user level of granularity, and expand upon ActiveX opt-in by allowing Opt-In functionality based on the user and the domain.

Traditionally, killbits have been used to effectively ban the instantiation of a control system-wide. This model is problematic in scenarios where a single user on a system of many users required the use of a particular control, but no others required it. Microsoft expanded upon killbits by introducing the registry key HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Ext\Settings\{CLSID}, where CLSID is the class id of the ActiveX control to restrict. By setting the Flags value of this key to "1", a control will be restricted for a single user. Figure 6 shows the Tabular Data Control disabled in this area of the registry.

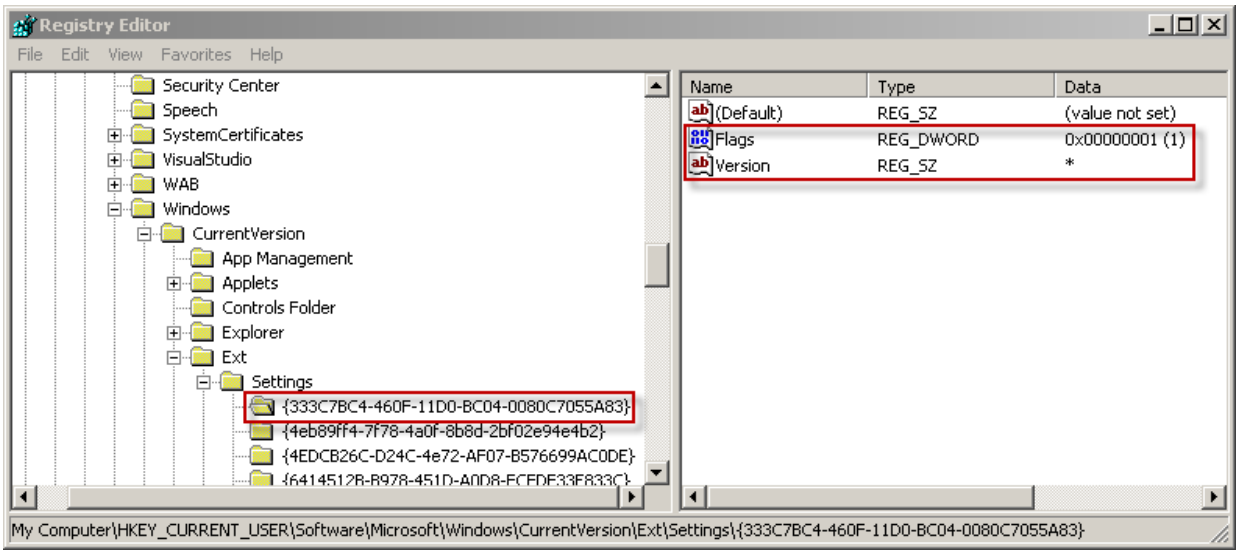


Figure 6: Example of a control restricted from a single user

Restricting ActiveX controls to certain domains allows the user to have more granular control over ActiveX security. Originally, SiteLock was the only method that allowed domain restriction, which was not configurable by the end user. This new per-domain restriction is managed in the registry by adding keys for specific allowed domains to HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Ext\Stats\{CLSID}\iexplore\AllowedDomains. A key for all domains can be added here by using the name "*", as opposed to a specific domain.

Per-Domain opt-in controls reduce the attack surface by requiring the user to approve the use of an ActiveX control before it is ran in the context of an unfamiliar domain. In effect, this would require an attacker to insert malicious web content onto a trusted domain in order to surreptitiously exploit the ActiveX control. Figure 7 shows the Tabular Data Control configured to run within the microsoft.com domain without prompting.

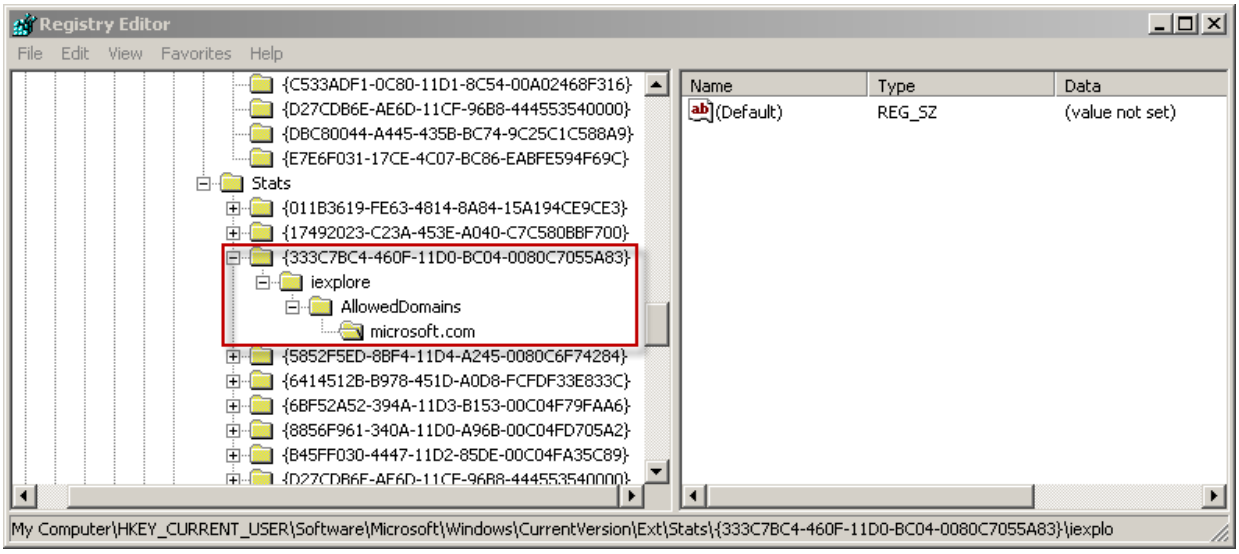


Figure 7: Example of a control approved to be run from the Microsoft.com domain

Internet Explorer Permission GUI

In addition to providing restriction capabilities, Microsoft enhanced the Internet Explorer UI by adding an interface that allows the user to easily configure ActiveX control permissions without having to modify the registry. Figure 8 shows how to access the Add-on Manager interface, and Figure 9 shows how to find DLLs that are allowed to run in the browser without permission.

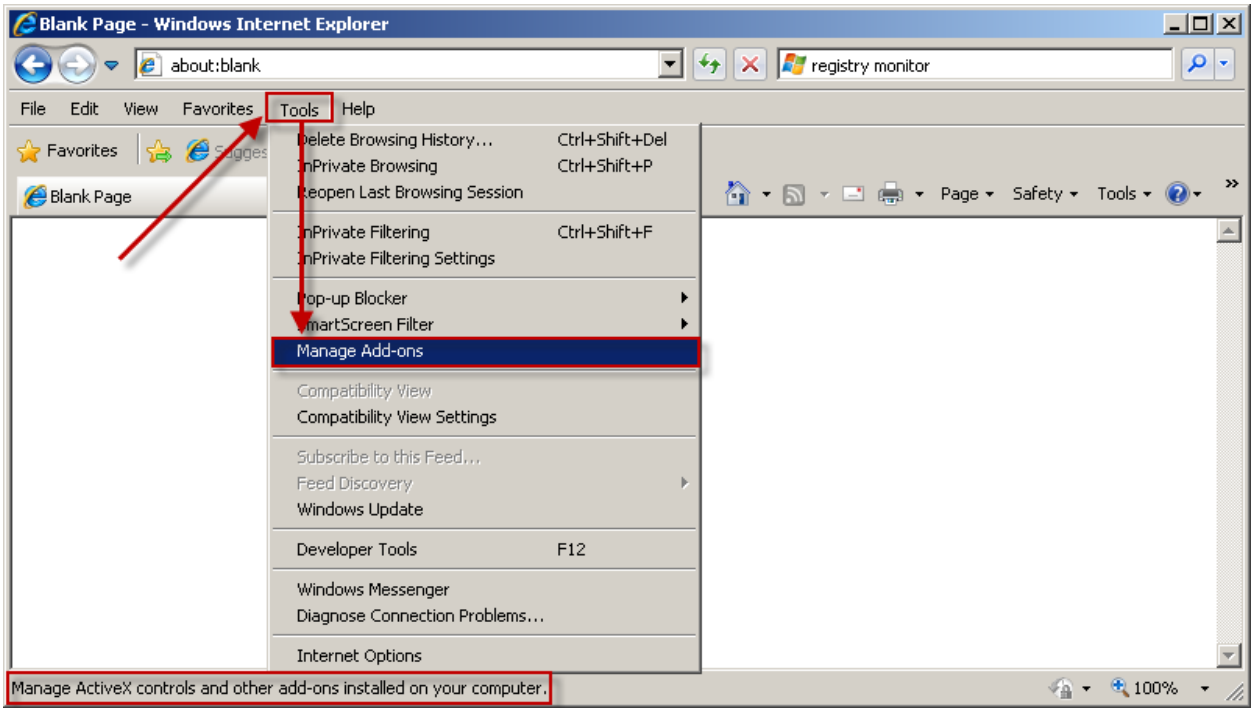


Figure 8: Navigating to the Add-on Manager

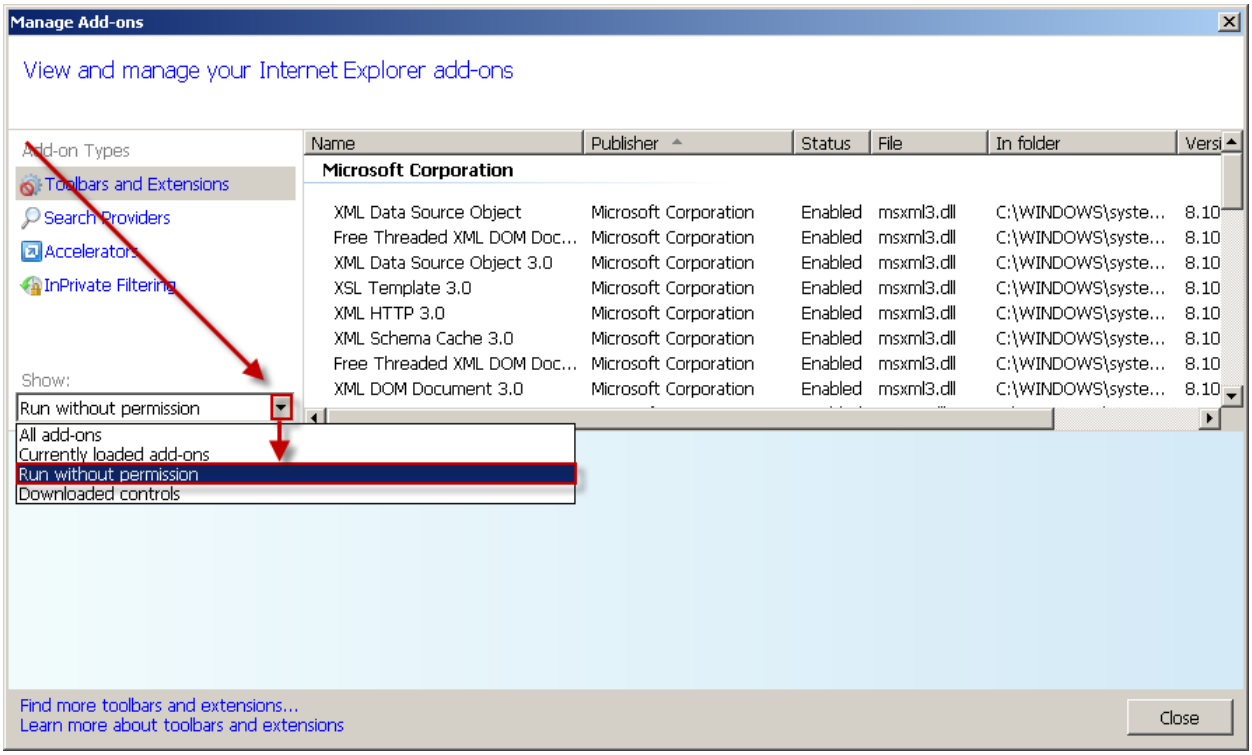


Figure 9: Operations to display controls that will run without permission

ActiveX Safety Wrap-Up

ActiveX has many methods to restrict which controls may load, and how they can be acted upon under a given context. One reason may very well be that, as interoperability has increased in applications, so too has opportunities for attackers. Under this premise, ActiveX security has evolved in an attack-response fashion and has led to a somewhat fractured security architecture. In later sections, we'll show an attack that allows some of these restrictions to be bypassed, mostly as a result of Microsoft adding security features to the browser in an ad-hoc fashion rather than having established a robust security architecture from the outset.

COM Overview

COM is an architectural standard that mandates a language agnostic representation of objects, and facilitates interaction between these objects. Microsoft uses COM as a fundamental building block in many of their premier technologies. It is pervasive in their flagship Windows Operating System, and also utilized extensively by many other peripheral products, such as Internet Explorer and Office. In the first section entitled Variants, we will discuss the fundamental, language agnostic data types that COM uses to communicate and the APIs used to manipulate them. Variants will be explored in order to provide the reader with more context for the types of vulnerabilities that this paper focuses on. Following variants is a section entitled COM Automation, which discusses the subset of COM objects that can be readily exposed to scripting runtime environments, collectively known as ActiveX controls. Finally, in the section entitled COM Persistence Overview, we will discuss the concept of persistence - the ability to serialize the current state of a COM object and subsequently resurrect that object at a later time. The use of persistence will be explored in the context of potentially hostile environments, where the serialized objects may originate from untrusted sources (such as malicious web pages or office documents).

Variants

VARIANTS are one of the key data structures utilized throughout the Windows platform for representing arbitrary data types in a standardized format. In particular, they are an integral part of COM, and are employed to exchange data between two or more communicating objects. The VARIANT data structure is a relatively simple one – it is composed of a type and a value, and is defined in OAIdl.h in the Windows SDK as shown.

```
struct __tagVARIANT
  {
  VARTYPE vt;
  WORD wReserved1;
  WORD wReserved2;
  WORD wReserved3;
  union
    {
    BYTE bVal;
    SHORT iVal;
    FLOAT fltVal;
    DOUBLE dblVal;
    VARIANT_BOOL boolVal;

                … more elements …

    BSTR bstrVal;
    IUnknown *punkVal;
    IDispatch *pdispVal;
    SAFEARRAY *parray;
```

```
        VARIANT_BOOL *pboolVal;
        _VARIANT_BOOL *pbool;
        SCODE *pscode;
        CY *pcyVal;
        DATE *pdate;
        BSTR *pbstrVal;
        VARIANT *pvarVal;
        PVOID byref;
    }   __VARIANT_NAME_1;
};
```

The value contained by a VARIANT can be one of a variety of different types, and so only has meaning when given context by the vt member, which indicates the type. There are quite a large number of basic types that can be represented by a VARIANT. Some of the more common ones are shown in Table 10.

Type Name	Value	Union Contains
VT_EMPTY	0x0000	Undefined
VT_NULL	0x0001	NULL value
VT_I2	0x0002	Signed (2-byte) short
VT_I4	0x0003	Signed (4-byte) integer
VT_R4	0x0004	Signed (4-byte) real (float)
VT_R8	0x0005	Signed large (8-byte) real (double)
VT_BSTR	0x0008	String; Pointer to a BSTR
VT_DISPATCH	0x0009	Pointer to an IDispatch interface (automation object)
VT_ERROR	0x000A	Error code (4-byte integer)
VT_BOOL	0x000B	Boolean (2-byte short)
VT_VARIANT	0x000C	Pointer to another VARIANT
VT_UNKNOWN	0x000D	Pointer to an IUnknown interface (any COM object)
VT_I1	0x0010	Signed (1-byte) char
VT_UI1	0x0011	Unsigned (1-byte) char
VT_UI2	0x0012	Unsigned (2-byte) short
VT_UI4	0x0013	Unsigned (4-byte) integer
VT_RECORD	0x0024	Pointer to an IRecordInfo interface (used to represent user-defined data types)

Table 10: VARIANT Basic Types

As can be seen in Table 10, all of the basic data types can be represented as a variant, in addition to a variety of COM interface types such as IUnknown and IDispatch interfaces. Furthermore, user-defined types are supported through the use of the IRecordInfo COM interface. This interface provides functions to define custom object sizes and marshallers so that any arbitrary data structure can be represented. The listed types are only a subset of all the supported VARIANT types. A complete list of all of the available types can be found in wtypes.h located within the Windows SDK.

In addition to basic variant types, there are several modifiers that, when used in conjunction with a basic type, alter the meaning of what is contained within the __VARIANT_NAME_1 union. Modifiers cannot be used on their own; they are specifically designed to provide additional context to a basic type. They are used by combining the modifier value (or values) with that of the basic type. The modifiers and their respective meanings are summarized in Table 11.

Modifier Name	Modifier Value	Value
VT_VECTOR	0x1000	Value points to a simple counted array (Rarely used)
VT_ARRAY	0x2000	Value points to a SAFEARRAY structure
VT_BYREF	0x4000	Value points to base type, instead of containing a literal of the base type

Table 11: VARIANT Modifier Types

As can be seen in the tables, basic types are all below 0x0FFF, and modifiers are single-bit values larger than 0x0FFF. So, by augmenting a basic type with a modifier using simple bit-masking operations, a new, complex type is formed. For example, a VARIANT containing an array of strings would have the type VT_ARRAY|VT_BSTR, and the value member would point to a SAFEARRAY where each member was a BSTR. (SAFEARRAYs will be examined in more depth momentarily.) A VARIANT could represent a pointer to a signed integer by having the type VT_BYREF|VT_I4. The VT_BYREF modifier may also be used in conjunction with one of the other modifiers, so a VARIANT could have the type (VT_BYREF|VT_ARRAY|VT_BSTR). In this case, the value member would point to a SAFEARRAY pointer, whose members are all of type BSTR.

Safe Arrays

Arrays are a common data construct utilized by COM, and are present in VARIANTs that contain the VT_ARRAY modifier in the vt field. In this case, a SAFEARRAY is used to encapsulate a series of elements of the same data type, and can be manipulated through the SafeArray API for safely accessing the members of the array without needing to worry about boundaries and other administrative problems associated with array access. Although they are most often used to represent an array with just a single dimension, SAFEARRAYs are also capable of representing multi-dimensional arrays of potentially differing dimension sizes (often referred to as “jagged arrays”). The SAFEARRAY structure definition is defined in OAIdl.h in the Windows SDK, and is shown below.

```
typedef struct tagSAFEARRAY
{
    USHORT cDims;
    USHORT fFeatures;
    ULONG cbElements;
    ULONG cLocks;
    PVOID pvData;
    SAFEARRAYBOUND rgsabound[ 1 ];
}  SAFEARRAY;
```

Elements contained within a SAFEARRAY are cbElements in size, and are stored contiguously in an area of memory, pointed to by the pvData member. An array of SAFEARRAYBOUND structures follows the SAFEARRAY descriptor in memory, with each SAFEARRAYBOUND structure describing a single dimension of the array. The SAFEARRAYBOUND structure is constructed as follows:

```
typedef struct tagSAFEARRAYBOUND
{
    ULONG cElements;
    LONG lLbound;
}     SAFEARRAYBOUND;
```

Simply put, the lLbound member indicates the lower bound of the described dimension, and the cElements member indicates how many members exist within that dimension.

The SAFEARRAY API is relatively extensive, so we will consider the most common API functions required for manipulation of these structures. The first two functions are for initialization and destruction, and are the complement of each other:

```
SAFEARRAY *SafeArrayCreate(VARTYPE vt, UINT cDims, SAFEARRAYBOUND *
      rgsabound);

HRESULT SafeArrayDestroy(SAFEARRAY * psa);
```

These functions are used to create and destroy an array respectively. When the array is created, the data type of each array member is designated, as well as the number of the dimensions of the array. These properties are both immutable; a SAFEARRAY's type and number of dimensions cannot be modified after creation.

There are two different ways of accessing data in arrays. The first way is to get a pointer to the memory where all of the elements reside, and is done using the following functions:

```
HRESULT SafeArrayAccessData(SAFEARRAY * psa, void HUGEP** ppvData);
HRESULT SafeArrayUnaccessData(SAFEARRAY * psa);
```

This is often the preferred method when accessing elements in a loop, in the form:

```
BSTR *pString;

if(FAILED(SafeArrayAccessData(psa, &pString))
      return -1;

for(i = 0; i < psa->rgsabound[0].cElements; i++)
{
      … operate on string …
}
```

```
SafeArrayUnaccessData(psa);
```

The second way to access data is by accessing an individual element using the following functions:

```
SafeArrayGetElement(SAFEARRAY * psa, LONG * rgIndices, void * pv);
SafeArrayPutElement(SAFEARRAY * psa, LONG * rgIndices, void * pv);
```

Each of these functions takes an array of indices and will either return or store the specific value in question. Note that internally, both functions verify the validity of the supplied indices to ensure that each array access is within bounds.

Lastly, we should mention that SAFEARRAYs have locking mechanisms to ensure exclusive thread access to array data, accessed by the following two functions:

```
HRESULT SafeArrayLock(SAFEARRAY * psa);
HRESULT SafeArrayUnlock(SAFEARRAY * psa);
```

VARIANT versus VARIANTARG

Many of the VARIANT API functions take either a VARIANT or a VARIANTARG. Microsoft documentation suggests that the difference between these two values is that VARIANTs always contain direct values (ie, they can't have the modifier VT_BYREF), whereas VARIANTARGs can. In fact, you will notice in the discussion of the VARIANT API further on that most of the Variant* functions take VARIANTARGs. In reality, these structures are actually equivalent and can be used interchangeably despite documentation indicating otherwise. Furthermore, a compiler error is not generated when they are used interchangeably. (Microsoft's documentation on their supposed distinctions is available at http://msdn.microsoft.com/en-us/library/ms221627.aspx.)

VARIANT API

The API for manipulating VARIANTs is quite extensive, however only a few of the functions are relevant for the purposes of this paper, and they are discussed in this section.

Variant Initialization and Destruction

VARIANTs are initialized using the VariantInit() function, which has the following prototype:

```
HRESULT VariantInit(VARIANTARG *pvarg);
```

This function does nothing except to set the type member of the VARIANT, vt, to VT_EMPTY, indicating that the VARIANT holds no value. The VARIANT is later cleaned up using the reciprocal function VariantClear():

```
HRESULT VariantClear(VARIANTARG *pvarg);
```

The VariantClear() function will also clear the vt member, as well as free any data associated with the VARIANT. For example, if the VARIANT contains an IDispatch or IUnknown interface (type VT_DISPATCH or VT_UNKNOWN respectively), then the interface will be released by VariantClear(). If the VARIANT is a string (VT_BSTR), it will be de-allocated, and so on.

Variant Manipulation

The two primary types of operations one may perform on a VARIANT using the API are conversion and duplication. There are a large variety of specific conversion functions of the form VarXXFromYY(), where XX is the destination VARIANT type and YY is the source type. There are also generic functions for converting between any two VARIANT types, which are shown below.

```
HRESULT VariantChangeType(VARIANTARG *pvargDest, VARIANTARG *pvargSrc,
      unsigned short wFlags, VARTYPE vt);
HRESULT VariantChangeTypeEx(VARIANTARG *pvargDest, VARIANTARG *pvargSrc, LCID
      lcid, unsigned short wFlags, VARTYPE vt);
```

These two functions both perform essentially the same task – converting pvargSrc to the type specified by vt, and placing the result in pvargDest. These functions will be revisited in further depth in Section 3 of this paper.

The other functions worth mentioning are those responsible for copying a VARIANT value from one VARIANT to another:

```
HRESULT VariantCopy(VARIANTARG *pvargDest, VARIANTARG *pvargSrc);
HRESULT VariantCopyInd(VARIANTARG *pvargDest, VARIANTARG *pvargSrc);
```

These functions both clear the destination VARIANT, and then copy in the source VARIANT. They do a deep copy; that is, if a COM interface is copied, the reference count is incremented, and so on. The difference between the two functions is that VariantCopyInd() will follow an indirect reference for a copy (ie. if the VARIANT has the VT_BYREF modifier, the value will be dereferenced and then modified), whereas VariantCopy() will not. VariantCopyInd() is also recursive; if a VARIANT is received that has the type (VT_BYREF|VT_VARIANT), the destination VARIANT will be examined further. If it is also a (VT_BYREF|VT_VARIANT), an error is signaled. If it has a VT_BYREF modifier but is not a VT_VARIANT, this VARIANT will be passed to VariantCopyInd() again, thus retrieving the value being stored.

COM Automation

As mentioned previously, COM Automation facilitates the integration of pluggable components into scripting environments. This is primarily achieved by creating objects that implement one or both of the automation interfaces: IDispatch and IDispatchEx. The IDispatch interface exposes functions that are designed to achieve the following directives:

1. Allow an object to be self-publishing – ie. Advertise its properties and methods
2. Allow methods to be called or properties to be manipulated by name, rather than direct VTable / memory manipulation.
3. Provide a unified marshalling interface for objects being passed to methods or properties, as well as objects being returned to the scripting host.

By implementing IDispatch, objects can be loaded at runtime by a host application and subsequently manipulated without the host having to know any compile time details about the objects it is. This capability is particularly useful for scripting interfaces which require extensibility.

The IDispatch interface is derived from IUnknown (both documented at MSDN), adding four methods as shown:

```
       /*** IDispatch methods ***/
    HRESULT (STDMETHODCALLTYPE *GetTypeInfoCount)(
        IDispatch* This,
        UINT* pctinfo);

    HRESULT (STDMETHODCALLTYPE *GetTypeInfo)(
        IDispatch* This,
        UINT iTInfo,
        LCID lcid,
        ITypeInfo** ppTInfo);

    HRESULT (STDMETHODCALLTYPE *GetIDsOfNames)(
        IDispatch* This,
        REFIID riid,
        LPOLESTR* rgszNames,
        UINT cNames,
        LCID lcid,
        DISPID* rgDispId);

    HRESULT (STDMETHODCALLTYPE *Invoke)(
        IDispatch* This,
        DISPID dispIdMember,
        REFIID riid,
        LCID lcid,
        WORD wFlags,
        DISPPARAMS* pDispParams,
        VARIANT* pVarResult,
        EXCEPINFO* pExcepInfo,
        UINT* puArgErr);
```

If an application would like to call any of the methods or modify any of the properties exposed by the object, it first needs to determine the dispatch ID associated with the method it would like to call. To determine this information, the application first needs to call GetIdsOfNames(). The return value is an integer that maps to the actual method that will be executed through the Invoke() method. The Invoke() method takes the ID of the member to be executed, the arguments to the method, and some other information about locale, etc as arguments. The wFlags argument passed to Invoke() defines whether the dispatch ID references a method exposed by the object or a property value that it should either get or set. The arguments to the method that will be executed are passed in a DISPPARAMS structure. The DISPPARAMS structure is defined below:

```
typedef struct FARSTRUCT tagDISPPARAMS{
VARIANTARG FAR* rgvarg;              // Array of arguments.
   DISPID FAR* rgdispidNamedArgs;    // Dispatch IDs of named arguments.
   Unsigned int cArgs;               // Number of arguments.
   Unsigned int cNamedArgs;          // Number of named arguments.
} DISPPARAMS;
```

As you can see, this structure passes the arguments to the method in an array of VARIANTs (See the section on VARIANTs for more detail). This array must be unmarshalled by the called method. In some cases, this can be a bit of a daunting task given the complexity of some of the VARIANT types that may be present in the array.

The IDispatch interface is useful for creating automation objects whose behavior is immutable - the properties and methods must be known at compile time and they don't change. However, in some cases, it is desirable to have objects whose behavior could be modified at runtime, and the IDispatchEx interface extends IDispatch to allow this additional functionality. With IDispatchEx objects, it is possible to add or remove properties or methods at runtime. This is functionality that is commonly required by more dynamic late-bound languages such as scripting languages (e.g. JavaScript).

The IDispatchEx is also derived from the IUnknown interface, adding eight methods as follows:

```
HRESULT DeleteMemberByDispID(
    DISPID id
);
HRESULT DeleteMemberByName(
   BSTR bstrName,
   DWORD grfdex
);
HRESULT GetDispID(
   BSTR bstrName,
   DWORD grfdex,
   DISPID *pid
);
HRESULT GetMemberName(
   DISPID id,
   BSTR *pbstrName
);
HRESULT GetMemberProperties(
   DISPID id,
   DWORD grfdexFetch,
   DWORD *pgrfdex
);
HRESULT GetNameSpaceParent(
   IUnknown **ppunk
);
HRESULT GetNextDispID(
   DWORD grfdex,
   DISPID id,
   DISPID *pid
);
HRESULT InvokeEx(
   DISPID id,
   LCID lcid,
   WORD wFlags,
   DISPARAMS *pdp,
   VARIANT *pVarRes,
   EXCEPINFO *pei,
   IServiceProvider *pspCaller
);
```

While there are some differences in the way dispatch IDs are retrieved, the main changes to IDispatchEx are those that allow for the creation and deletion of object properties and methods. GetDispID(), for example, differs from GetIdsOfNames() in that it can be told to create a new name and dispatch ID for a new property or method. Additionally, you can see the methods DeleteMemberByName() and DeleteMemberByDispID() have been added. In ActiveX controls that extend the IDispatchEx interface, the dynamic creation and deletion of members is accessible through JavaScript.

Interestingly, JavaScript (for Internet Explorer) itself is implemented using a modified IDispatchEx interface exposed by the Microsoft script engine. Conceptually, this implementation makes sense because JavaScript will need to be able to create objects and add and delete members all without any preconceived notion of what the object may look like. So, for example, when JavaScript creates a new object:

```
Obj = new Object();
```

Internet Explorer will first call the GetDispID() method for Obj – ensuring the fdexNameEnsure flag is set to create the member. It will then call its own internal version of Invoke() to call the Object() method. The value returned from the call to Invoke() will then become assigned to the Obj member.

COM Persistence Overview

COM provides two primary interfaces for manipulating an object's persistence data. The first interface, IStream, represents a data stream that is used to store a single object's persisted data. It supports standard file operations including reading, writing, and seeking using the interface methods. The IStream interface abstracts the underlying storage details from the consumer of the stream. This abstraction allows for COM objects to implement serialization functionality without explicit knowledge of the underlying backing store. This abstraction is visually depicted in Figure 12.

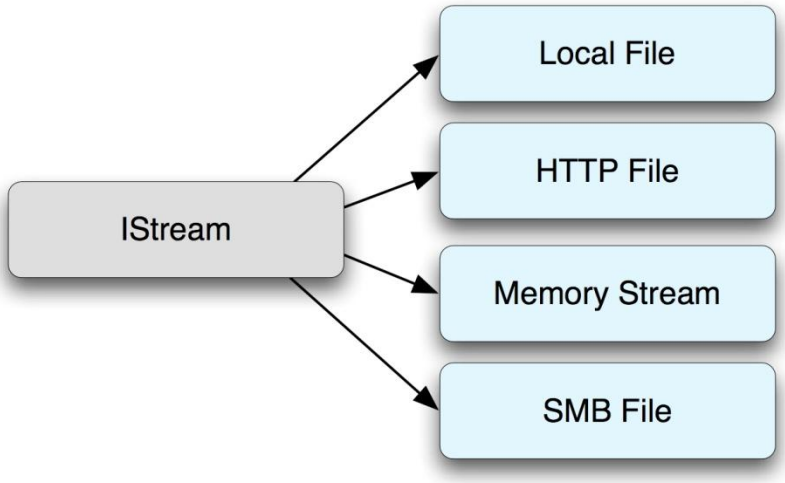


Figure 12: Diagram representing various media that can contain the IStream data.

The second interface, IStorage, is employed when a program or COM object requires the persistence of multiple objects. IStorage represents a storage file, which can hold logically separate binary streams inside a single file using unique names to identify each stream. Additionally, a storage file can contain logically separate subordinate storage files, also accessed by unique names, thus allowing for recursion if it is required. The IStorage interface provides methods that allow the programmer to access each of the constituent streams and subordinate storage files. Figure 13 depicts an example of a typical storage file.

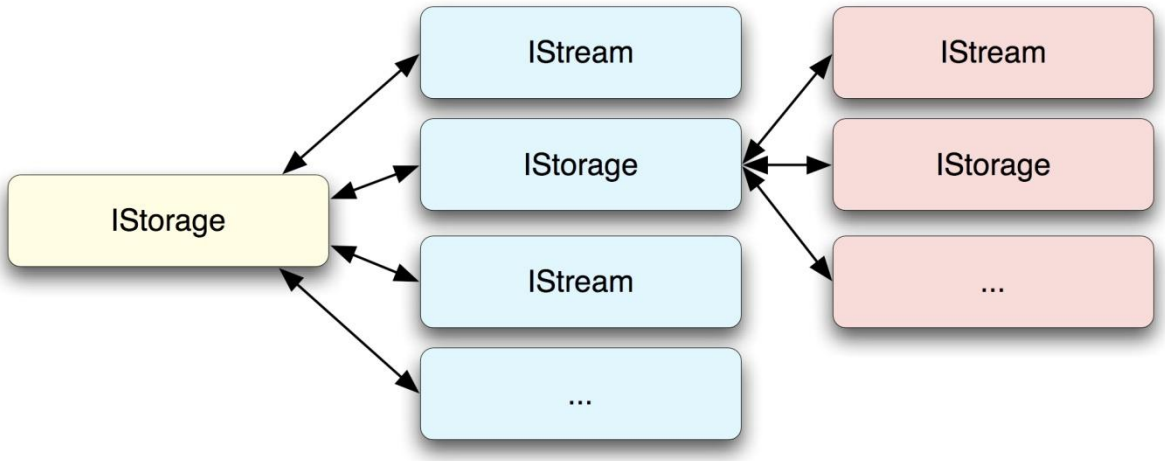


Figure 13: An example of a storage file's contents.

In addition to IStream and IStorage, there are several other interfaces that can be used for manipulating COM persistence data, depending on the medium that contains the data. The following is a list of interfaces that can store persistent object data.

- IMoniker
- IFile
- IPropertyBag
- IPropertyBag2

COM objects support serialization by implementing one of several well-known persistence interfaces. Each of these persistence interfaces are specializations of the IPersist interface, which has the following definition:

```
    MIDL_INTERFACE("0000010c-0000-0000-C000-000000000046")
    IPersist : public IUnknown
    {
    public:
        virtual HRESULT STDMETHODCALLTYPE GetClassID(
       /* [out] */ __RPC__out CLSID *pClassID) = 0;
```

```
    };
```

Each subclass of IPersist has methods named Load() and Save(), which serialize the data and resurrect the data, respectively. The differentiator between these subclasses is the type of interface that holds the persisted data. Table 14 lists the persistence interfaces, and the argument type that each respective interface uses to hold the data. Figure 15 visually depicts the inheritance hierarchy of these interfaces.

Persistence Interface	Argument that Holds the Data
IPersistFile	An LPCOLESTR that designates a standard file path
IPersistMemory	An LPVOID that is a fixed-size memory buffer
IPersistMoniker	An IMoniker interface
IPersistPropertyBag	An IPropertyBag interface
IPersistPropertyBag2	An IPropertyBag2 interface
IPersistStorage	An IStorage interface
IPersistStream	An IStream interface
IPersistStreamInit	An LPSTREAM interface

Table 14: Persistence Interfaces correlated to Data Interfaces

When a host program wishes to serialize an object, it will query that object for a persistence interface. If successful, the application will then call the Save() method, passing a pointer to one of the previously-discussed storage interfaces (IStream, IStorage, IFile, etc). Later, when a host program wishes to resurrect the object from its persistent state, it will once again retrieve the object's persistence interface, and call the Load() method. The object resurrected from the persistence data should be equivalent to the object that was previously saved.

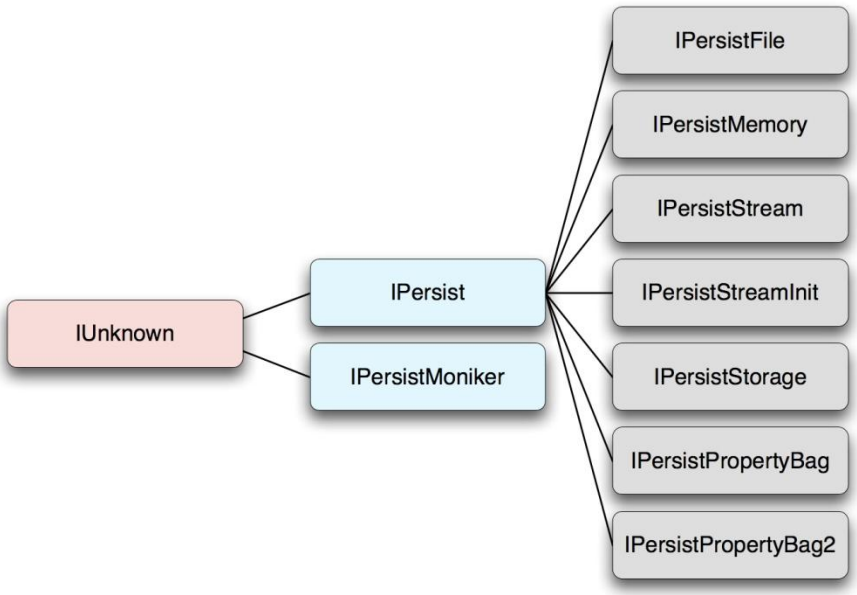


Figure 15: The inheritance hierarchy of persistence interfaces.

Implementing COM Persistence in the ATL

Developers of COM objects are free to implement their own persistence interfaces. If these developers choose to write their own code for the interface, they would manipulate the interface that stores the persistence data by reading and writing data in an arbitrary format. However, most developers choose to use template classes provided in the Microsoft ATL, when there is template code to do so, avoiding the extra work it would require to implement these interfaces. Version nine of the Microsoft ATL has template classes for the following persistence interfaces.

- IPersist
- IPersistPropertyBag
- IPersistStorage
- IPersistStreamInit

The template code requires a programmer to define a series of properties, known as a property map, which the persistence interface will use as a boiler plate for serializing and resurrecting the object in question. This property map is a terminated array of structures that list the properties for the control that must be serialized and resurrected, and should be made explicit enough to guarantee that the object, once serialized, will be equivalent to an object that is resurrected from the data. Version nine of the ATL includes various macros to aid a programmer when defining these properties and include macros from the following list.

- BEGIN_PROPERTY_MAP
- BEGIN_PROP_MAP
- PROP_ENTRY
- PROP_ENTRY_EX
- PROP_ENTRY_TYPE
- PROP_ENTRY_TYPE_EX
- PROP_PAGE
- PROP_DATA_ENTRY
- END_PROPERTY_MAP
- END_PROP_MAP

Each of the previously mentioned macro functions take various arguments and use them to define an ATL_PROPMAP_ENTRY structure. The following code is the structure definition taken from version nine of the ATL.

```
struct ATL_PROPMAP_ENTRY
{
        LPCOLESTR szDesc;
        DISPID dispid;
        const CLSID* pclsidPropPage;
        const IID* piidDispatch;
        DWORD dwOffsetData;
        DWORD dwSizeData;
        VARTYPE vt;
};
```

The elements in the ATL_PROPMAP_ENTRY structure are all quite important to understand, and are summarized in Table 16.

Element Name	Element Purpose
szDesc	Unicode string that uniquely identifies the property name
dispid	32-bit integer that uniquely identifies the property within the object
pclsidPropPage	Pointer to a COM class id that identifies a COM class that offers a GUI interface to set and retrieve the property within the control.
piidDispatch	Pointer to a COM interface id that describes an interface that inherits from IDispatch, which can be used to set the property through the Invoke method of the interface
dwOffsetData	32-bit value that specifies the property's memory offset from the beginning of the object
dwSizeData	32-bit value that specifies the number of bytes that have been allocated in the object to hold the property's data
vt	16-bit value that specifies the property's type

Table 16: A listing of the elements of the ATL_PROPMAP_ENTRY structure and the purpose they serve.

The macro functions for defining properties use arguments supplied to the function to set certain ATL_PROPMAP_ENTRY elements, and will set others to a default state. Depending on the elements that have non-default values, the template code responsible for the persistence operations will use slightly differing strategies when serializing and resurrecting the data. Both BEGIN_PROPERTY_MAP and

BEGIN_PROP_MAP will include code that starts to define the structure; however, the former will automatically include X and Y position information within the property map. END_PROP_MAP and END_PROPERTY_MAP are macro functions that will include a terminating ATL_PROPMAP_ENTRY element and end the structure definition. Between BEGIN_PROPERTY_MAP or BEGIN_PROP_MAP, and END_PROP_MAP or END_PROPERTY_MAP, are ATL_PROPMAP_ENTRY instances that describe the properties of a COM object.

PROP_ENTRY and PROP_ENTRY_EX both define a property using the property's name, display id, and a property page that can be used to set the property. PROP_ENTRY_TYPE and PROP_ENTRY_TYPE_EX define the same information as PROP_ENTRY and PROP_ENTRY_EX; however they also require an explicit variant type that is expected when dealing with the property. The "_EX" suffix designates that the macro function also expects an explicit dispatch interface id that should be used when setting or getting the property's value. The PROP_DATA_ENTRY macro requires a unique string identifier for the property, the name of the class's member that will be used to store the property, and the type of variant that's expected for the property. Internally, the PROP_DATA_ENTRY macro uses the offsetof and sizeof structure to explicitly define dwOffsetData and dwSizeData within the ATL_PROPMAP_ENTRY structure. PROP_PAGE is used to specify a COM class id that offers a GUI interface, which can manipulate the properties of an object.

To help illustrate the use of property maps in C code and how properties are read from a persisted state, we'll briefly present an example COM object called HelloCom. HelloCom is a simple ActiveX control that can store a person's first and last names. The properties will have the following names:

- NameFirst
- NameLast

The following C++ code snippet shows portions of code for the HelloCom control that are relevant for implementing persistence.

```
class HelloCom :
        public  IPersistStreamInitImpl<HelloCom>,
        public IPersistStorageImpl<HelloCom>,
        public IPersistPropertyBagImpl<HelloCom>,
 {
public:
BEGIN_PROP_MAP(HelloCom)
       PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
       PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
       PROP_ENTRY("NameFirst", 1, CLSID_HelloComCtrl)
       PROP_ENTRY_TYPE("NameLast", 2, CLSID_HelloComCtrl, VT_BSTR)
END_PROP_MAP()
};
```

If the application is loading the persistence data from a binary stream, then the application would query for the IPersistStreamInit interface and would receive a vtable pointing to the IPersistStreamInitImpl template class. Next, the application will call the Load() method, passing in an IStream object that will be used to read the persistence data. Prior to any of the serialized data in a stream, a version number is stored in order to deal with backwards compatibility issues. So, the first four bytes in the stream will be a little-endian representation of the ATL version that was used to compile the control. In Visual Studio 2008, this value is 0x00000900. As long as the value is less-than or equal-to the version of the ATL used to compile the control, processing can resume, otherwise, an error is signaled.

After the versioning information has been processed, the properties themselves can then be retrieved from the stream. The bytes immediately after the version number in the stream in this case would be two 4-byte little-endian representations of the _cx and _cy elements. Since these elements were declared with the PROP_DATA_ENTRY macro, these 32-bit values will be written directly to the memory offset in the class where the `m_sizeExtent.cx` and `m_sizeExtent.cy` values reside.

Following these values, we will encounter the serialized representation of NameFirst. Since NameFirst is declared in the property map using the PROP_ENTRY() macro, which contains no data type, the type information needs to be retrieved from the stream. Therefore, the first two bytes in the stream would be an unsigned 16-bit value of 0x0008, representing the variant type VT_BSTR. Next would come an unsigned 32-bit value specifying the length of the string. If the name were "Example", then the value of this 32-bit integer specifying the size would equal 0x10; seven 2-byte characters plus a terminating null. The next values would be the characters that represent the name, followed by a terminating 16-bit value of 0x0000. NameLast would come next, and would be specified identically to NameFirst, except that the 16-bit variant type specifier would be absent in the stream, since the type is explicitly declared in the property map using the PROP_ENTRY_TYPE() macro.

Table 17 shows an example of the stream described in the previous paragraphs, with hexadecimal values representing the value in the stream, an offset showing the position of the value in the stream, and a description of how the values should be interpreted.

Offset	Hexadecimal representation of bytes	Description
0x00	00 09 00 00	Version nine of the ATL
0x04	00 01 00 00	The _cx value is 256
0x08	00 01 00 00	The _cy value is 256
0x0C	08 00	NameFirst is stored as a VT_BSTR
0x0E	0C 00 00 00	NameFirst is 12 characters long
0x12	46 00 69 00 72 00 73 00 74 00 00 00	NameFirst is equivalent to "First"
0x1E	0A 00 00 00	NameLast is 10 bytes long
0x22	4C 00 61 00 73 00 74 00 00 00	NameLast is equivalent to "Last"

Table 17: A listing of the elements contained in a stream for the fictitious HelloCom example

COM Persistence in Microsoft Internet Explorer

Microsoft Internet Explorer uses persistence when assigning values to properties of ActiveX objects. The six main interfaces used by Internet Explorer, ordered by preference, are IPersistPropertyBag, IPersistMoniker, IPersistFile, IPersistStreamInit, IPersistStream, and IPersistStorage. The browser will attempt to retrieve an interface pointer to each persistence interface in sequence until it is successful, or no interfaces have been found, at which point the operation fails.

The first, and most familiar, persistence interface is IPersistPropertyBag. IPersistPropertyBag has been specifically designed to allow persistent objects to be embedded within HTML. Take, as an example, the following HTML code that embeds Microsoft Media Player within a web page.

```
<OBJECT id="VIDEO" CLASSID="CLSID:6BF52A52-394A-11d3-B153-00C04F79FAA6" >
        <PARAM NAME="URL" VALUE="MyVideo.wmv">
        <PARAM NAME="enabled" VALUE="True">
        <PARAM NAME="AutoStart" VALUE="False">
        <PARAM name="PlayCount" value="3">
        <PARAM name="Volume" value="50">
        <PARAM NAME="balance" VALUE="0">
        <PARAM NAME="Rate" VALUE="1.0">
        <PARAM NAME="Mute" VALUE="False">
        <PARAM NAME="fullScreen" VALUE="False">
        <PARAM name="uiMode" value="full">
</OBJECT>
```

The <PARAM> tags that appear within the <OBJECT> tag represent the COM object's property names and persisted values. When Internet Explorer parses a web page and encounters these PARAM tags, it first creates a PropertyBag class and queries for the IPropertyBag interface. Next, it will parse the name and value parameters of the PARAM html tag and call the Write() method on the IPropertyBag interface, supplying the name and a string representation of the value for the property it has parsed. Once Internet Explorer has loaded all of the PARAM tags into a property bag, it will query the COM object (In the above example, a Media Player object) for an IPersistPropertyBag interface. Internet Explorer will then call the Load() method of the IPersistPropertyBag interface, passing the PropertyBag that was parsed from the HTML. The Load() method of the COM object will then convert the properties from a string representation into the object's preferred representation, and subsequently save the converted representation within the COM object. This strategy is employed by Internet Explorer to resurrect the object from a persistent state when it encounters the above HTML.

The reciprocal operation to the resurrection operation, serialization, is most commonly encountered when using the innerHTML attribute of an object. Consider the following JavaScript code, used in the same web page as the above HTML.

```
<script language="JavaScript">
```

```
        alert(VIDEO.innerHTML);
</script>
```

Upon processing the previous JavaScript, the web page will alert the user with a message box with HTML formatted text similar to the following example:

```
<PARAM NAME="URL" VALUE="./MyVideo.wmv">
<PARAM NAME="rate" VALUE="1">
<PARAM NAME="balance" VALUE="0">
<PARAM NAME="currentPosition" VALUE="0">
<PARAM NAME="defaultFrame" VALUE="">
<PARAM NAME="playCount" VALUE="3">
<PARAM NAME="autoStart" VALUE="0">
<PARAM NAME="_cx" VALUE="6482">
<PARAM NAME="_cy" VALUE="6350">
```

When Internet Explorer serializes an object using a PropertyBag, it first creates an instance of the PropertyBag class. Next, it queries the object to be persisted for the IPersistPropertyBag interface. Once the interface is retrieved, Internet Explorer calls the Save() method, passing the PropertyBag class instance. Finally, Internet Explorer will serialize the PropertyBag class into a format that is compatible with HTML standards.

The second, less common, way of inserting persistence data into a control over Internet Explorer is through the use of the data parameter of the OBJECT tag. An example of this type of persistence is shown in the following HTML.

```
<OBJECT
        id="VIDEO"
        CLASSID="CLSID:6BF52A52-394A-11d3-B153-00C04F79FAA6"
        data="./persistence_data"
        type="application/x-oleobject"
/ >
```

In the example above, instead of using PARAM tags, the persistence data is communicated through the data parameter of the object tag. When Internet Explorer encounters an object tag in this format, it follows a complex strategy to resurrect the object from the serialized data.

Internet Explorer will first check the file name specified in the data parameter to see if the file name extension is equal to ".ica", ".stm", or ".ods". If the extension is one of these, then it creates an IStream that can read binary data from the supplied file URL. Internet Explorer will then create an instance of the object specified in the first sixteen bytes of the file, or, if those sixteen bytes are zero, the CLASSID parameter in the object tag and query for the IPersistStream interface. If the interface is successfully retrieved, Internet Explorer will then call the Load() method of the interface, passing in the IStream. Next, the COM object will parse the stream and convert the binary data into the preferred representation of each property. Once these operations are finished, Internet Explorer will have a fully resurrected COM object.

If the filename does not match one of the well-known extensions, Internet Explorer does some extra work to determine what type of persistence interface to use for the COM object and corresponding persistence data. First, Internet Explorer will query the COM object for an IPersistFile interface. If the interface is successfully retrieved, it will call the Load() method of the COM object's interface, passing in a file path. It is then the COM object's responsibility to open the file and parse the data.

If the object does not support the IPersistFile interface, Internet Explorer will use the URL in the data value, and create an IStream object. Next, it will query the COM object for an IPersistStreamInit interface. If this operation is successful, then Internet Explorer will call the Load() method of the IPersistStreamInit interface, passing in the IStream object. If the COM object doesn't support the IPersistStreamInit interface, it will then attempt to query the object for an IPersistStream interface. If the object implements this interface, then Internet Explorer will call the Load() method of the IPersistStream interface, passing in the IStream object. If these operations are successful, then the COM object's IPersistStreamInit or IPersistStream interface is charged with resurrecting the properties from the given persistence data.

If the COM object does not implement either IPersistStreamInit, or IPersistStream, or the Load() method returns with a failure, then Internet Explorer will attempt to load the URL as a compound OLE document by calling StgOpenStorage from OLE32. If StgOpenStorage returns with a successful value, then Internet Explorer will query the COM object for an IPersistStorage interface. If the COM object indeed implements the IPersistStorage interface, then Internet Explorer will call the Load() method of the interface, passing in an IStorage object. From here it is, again, the responsibility of the COM object to parse the data contained in the IStorage object.

Attack Surface

The attack surface for COM can be divided into three areas. These areas are as follows:

- Methods exposed by objects within the browser
- Serialization of COM objects, and
- Marshalling values between components of the web browser

The first attack surface is really one that has been addressed many times before. Indeed, there are numerous speeches centered on targeting ActiveX controls as well as tools developed to automatically fuzz test exposed methods for vulnerabilities. (For interested readers, a paper about ActiveX fuzzing written by Will Dormann and Dan Plakosh from CERT was released recently and is available along with a fuzzing tool, available at http://www.cert.org/archive/pdf/dranzer.pdf. Another popular ActiveX fuzzer, AxMan, was released by HD Moore, available at http://www.metasploit.com/users/hdm/tools/axman/.)

The serialization of COM objects, also known as persistence, is another area that has been largely under-explored for security problems. We will examine the security implications of persistence quite extensively throughout Section 3, discussing de-serialization issues, type confusion vulnerabilities as a result of persistent objects, and trust boundaries that can be breached through object instantiation.

Finally, in Section 3, we will examine marshalling code in the context of security. This is another major attack surface that has been largely unexplored, most probably due to its implicit nature. Charged with leveraging a sometimes unintuitive API to keep track of memory allocations, object usage, and type conversions in an abstract manner, marshalling code can be quite difficult to write. We intend to discuss the types of issues that often occur when performing some level of marshalling. We will consider popular APIs and interfaces, as well as speak on a more general level about classes of problems that are more prevalent in marshalling code than anywhere else.

NPAPI Plugins

The Netscape Plugin Application Programming Interface (NPAPI) is the premier plugin architecture adopted by many contemporary web-browsers including Mozilla Firefox, Google Chrome, Apple Safari, and Opera. The architecture provides a simple model for creating plugins that expose functionality to the web browser via defined API calls. Although NPAPI is somewhat limited in its original incarnations, major revisions over time have allowed the creation of plugins that can not only process specialized objects embedded in a web page, but also expose them to scripted control from hosted scripting languages such as JavaScript. This is primarily due to the collaborative effort of several companies (Mozilla, Apple, Macromedia, Opera, and Sun) in 2004 to extend the NPAPI by adding the so-called NPRuntime, which provides a cross-platform standard for exposing objects to the browser DOM. This section is aimed at providing technical details for how the NPAPI is utilized; specifically focusing on the NPRuntime component, as this component is the most relevant feature that will be discussed in the following sections of this paper.

Plugin Registration

Before delving into the details of the NPAPI, we will briefly explore the process by which plugins are registered to the browser. This knowledge is required for being able to enumerate the attack surface of a given installation.

Plugins are shared libraries, at the most simple level, that are registered with the browser and are designed to handle specialized object types. When they are registered, the objects the plugins handle are specified in the form of MIME types, file extensions, or a combination of the two. The way in which plugins are registered and associated with MIME types/extensions differs depending on the browser and the platform. This section considers Windows installations of Mozilla Firefox, but analogous processes are available in other environments.

Plugins are registered to the Firefox browser in one of two ways:

1. They are copied into the plugins directory of the browser (typically C:\Program Files\Mozilla Firefox\plugins), or
2. A key is added to the registry to indicate the location and other specifics of the plugin (either in HKEY_LOCAL_MACHINE\Software\MozillaPlugins or HKEY_CURRENT_USER\Software\MozillaPlugins. The structure of the various subkeys required for a plugin is documented at https://developer.mozilla.org/en/Plugins/The_First_Install_Problem.)

Information for associated MIME types and file extensions for a given plugin is located in the Version information within the compiled DLL. The MIME types are specified in a string of pipe-seperated ('|') MIME identifiers like so:

```
MIMEType:   mime/type-1|mime/type-2|mime/type-3
```

Similarly, file extensions are also organized in a pipe-separated list as shown:

```
FileExtents:      ext1|ext2|ext3
```

A quick way of being able to examine the available plugins for a given Firefox installation is to simply browse to the URL about:plugins, which provides a list of the available installed plugins as well as the MIME types and file extensions associated with each one.

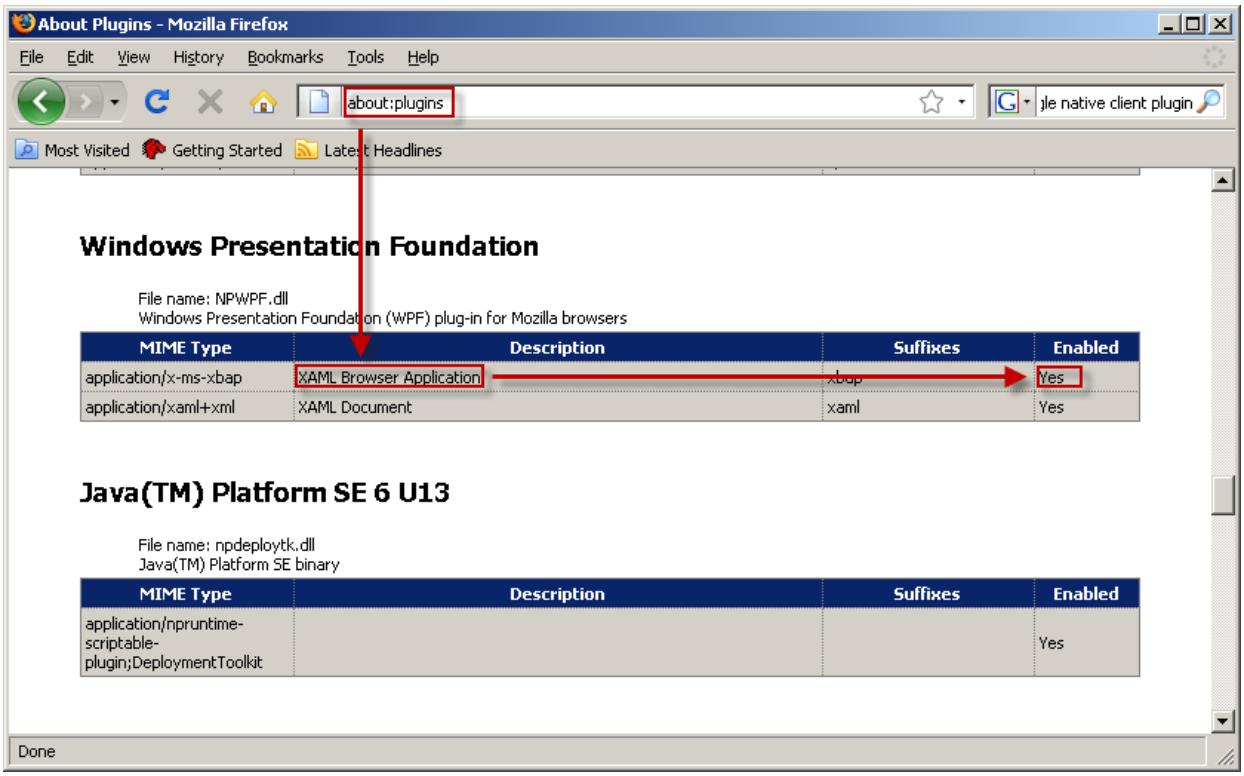


Figure 18: A screen shot that shows how to enumerate which plugins are enabled

The NPAPI and Plugin Initialization

The NPAPI is roughly broken down into two sets of functions: Browser-side functions and plugin-side functions. The browser-side functions represent the API that is exported by the browser to the plugins. This browser side API is contained in a structure named NPNetscapeFuncs, which is defined in npupp.h in the NPAPI SDK (available as part of the Gecko SDK: https://developer.mozilla.org/En/Gecko_SDK), and is shown below.

```
typedef struct _NPNetscapeFuncs {
    uint16 size;
    uint16 version;
    NPN_GetURLUPP geturl;
    NPN_PostURLUPP posturl;
    NPN_RequestReadUPP requestread;
    NPN_NewStreamUPP newstream;
    NPN_WriteUPP write;
    NPN_DestroyStreamUPP destroystream;
    NPN_StatusUPP status;
    NPN_UserAgentUPP uagent;
    NPN_MemAllocUPP memalloc;
    NPN_MemFreeUPP memfree;
    NPN_MemFlushUPP memflush;
    NPN_ReloadPluginsUPP reloadplugins;
    NPN_GetJavaEnvUPP getJavaEnv;
    NPN_GetJavaPeerUPP getJavaPeer;
```

```
    NPN_GetURLNotifyUPP geturlnotify;
    NPN_PostURLNotifyUPP posturlnotify;
    NPN_GetValueUPP getvalue;
    NPN_SetValueUPP setvalue;
    NPN_InvalidateRectUPP invalidaterect;
    NPN_InvalidateRegionUPP invalidateregion;
    NPN_ForceRedrawUPP forceredraw;
    NPN_GetStringIdentifierUPP getstringidentifier;
    NPN_GetStringIdentifiersUPP getstringidentifiers;
    NPN_GetIntIdentifierUPP getintidentifier;
    NPN_IdentifierIsStringUPP identifierisstring;
    NPN_UTF8FromIdentifierUPP utf8fromidentifier;
    NPN_IntFromIdentifierUPP intfromidentifier;
    NPN_CreateObjectUPP createobject;
    NPN_RetainObjectUPP retainobject;
    NPN_ReleaseObjectUPP releaseobject;
    NPN_InvokeUPP invoke;
    NPN_InvokeDefaultUPP invokeDefault;
    NPN_EvaluateUPP evaluate;
    NPN_GetPropertyUPP getproperty;
    NPN_SetPropertyUPP setproperty;
    NPN_RemovePropertyUPP removeproperty;
    NPN_HasPropertyUPP hasproperty;
    NPN_HasMethodUPP hasmethod;
    NPN_ReleaseVariantValueUPP releasevariantvalue;
    NPN_SetExceptionUPP setexception;
    NPN_PushPopupsEnabledStateUPP pushpopupsenabledstate;
    NPN_PopPopupsEnabledStateUPP poppopupsenabledstate;
} NPNetscapeFuncs;
```

When a plugin is initially loaded into memory, it is initialized by calling the function NP_Initialize(), which the plugin is required to export. The NPNetscapeFuncs structure is passed as the first parameter to this function by the browser, thereby exposing its API to the plugin. The reader should note that the information present in the size and version elements allow for extensions to the API, which indeed has been put into use. The SDK encourages the prefix NPN_* for browser side functions (“Netscape Plugin: Navigator”), so the rest of this paper will refer to callbacks using that convention.

The plugin-side functions are those that the plugin implements and are collectively used to define the plugin's functionality. The plugin-side functions are contained within the `NPPluginFuncs` structure, which is also defined in npupp.h, and is shown.

```
   typedef struct _NPPluginFuncs {
    uint16 size;
    uint16 version;
    NPP_NewUPP newp;
    NPP_DestroyUPP destroy;
    NPP_SetWindowUPP setwindow;
    NPP_NewStreamUPP newstream;
    NPP_DestroyStreamUPP destroystream;
    NPP_StreamAsFileUPP asfile;
    NPP_WriteReadyUPP writeready;
    NPP_WriteUPP write;
    NPP_PrintUPP print;
    NPP_HandleEventUPP event;
    NPP_URLNotifyUPP urlnotify;
    JRIGlobalRef javaClass;
    NPP_GetValueUPP getvalue;
    NPP_SetValueUPP setvalue;
} NPPluginFuncs;
```

Plugins are required to expore the NP_GetEntryPoints() function, which uses the NPPluginFuncs structure to communicate plugin information to the browser when the plugin is being initialized. The browser calls NP_GetEntryPoints, passing a pointer to a memory location that can hold the NPPluginFuncs structure. In turn, NP_GetEntryPoints populates the structure with the information for the plugin. By convention, plugin function names are prefixed with NPP_* (Netscape Plugin: Plugin), and we will try to adhere to this convention throughout this paper.

Plugin Initialization and Destruction

NPAPI plugins have two levels of initialization – the first, which we have already seen, is the one-time initialization performed when the browser loads the plugin. As we previously noted, this loading is achieved by calling the exported function NP_Initialize(). There is also instance initialization, which occurs each time the plugin is instantiated. For example, if the same plugin is utilized in two different <OBJECT> tags on the same page, a one-time load initialization will be performed, followed by two instance initializations. The instance initialization is performed by the plugin's NPP_New() function, which is defined as follows:

```
NPError NPP_New(NPMIMEType pluginType, NPP instance, uint16 mode, int16 argc,
      char *argn[], char *argv[], NPSavedData *saved);
```

There are quite a few parameters to this function that provide the plugin with instance information to aid the initialization process. The pluginType parameter denotes the MIME type that was associated with this instance of the plugin. Many plugins register several MIME types, so this parameter allows each instance to distinguish the MIME type it is supposed to be handling. The second parameter, instance, is a pointer to an instance of the plugin object, which has a pdata member that the plugin may use to save any private data specific to the current plugin instance. It is common practice for plugins to save a C++ object here. The next parameter is a mode argument which may take the values NP_EMBED(1) to indicate that the object is embedded in a web page, or NP_FULL (2) if the plugin represents a full-page object. The next three parameters are related to the <PARAM> values supplied to the object (or attributes within the <EMBED> tag, if it was used instead of <OBJECT>). The argc argument indicates the quantity of parameters that were supplied in the argn and argv arrays. The two string arrays argn, and argv have an element count equal to the argc argument, and specify the parameter names and values, respectively. Finally, the saved argument can be used to access data that was saved by a previous instance of the plugin using NPP_Destroy(), a function we will explore momentarily.

Destruction occurs by the reciprocal function NPP_Destroy(), which has the following definition:

```
NPError NPP_Destroy(NPP instance, NPSavedData **saved);
```

This function simply takes an instance pointer and an NPSavedData **, which can be used to retain information for the next plugin instance, as described previously.

Streams

Streams are also quite relevant to the attack surface of a typical NPAPI plugin. A stream object, represented by the NPStream data structure, represents an opaque data stream that is either sent from the browser to the plugin, or vice versa. Plugin instances can deal with multiple streams, but each stream is specific to that plugin instance; they cannot be shared.

New streams are sent from the browser to the plugin by calling the plugin-side function NPP_NewStream(), which has the following prototype.

```
NPError NPP_NewStream(NPP instance, NPMIMEType type, NPStream *stream, NPBool
seekable, uint16 *stype);
```

Most of these parameters are self-explanatory, except for the stype parameter, which the plugin fills out as one of the following values:

- NP_NORMAL (1) – Stream data is delivered as it is pulled. This is the default mode of operation.
- NP_ASFILEONLY (2) – Data is saved locally in a temporary file first
- NP_ASFILE (3) – Data is delivered normally as with NP_NORMAL, but is also saved to a temporary file
- NP_SEEK (4) – Stream data can be accessed randomly as opposed to sequentially.

Streams are delivered to the plugin when either a file is associated with the plugin instance (such as SWF files for the Adobe Flash plugin), or when the plugin requests a stream by calling the NPN_GetURL(), NPN_GetURLNotify(), NPN_PostURL(), or NPN_PostURLNotify() functions.

Streams are later destroyed by calling the NPP_DestroyStream() function. This function has the following prototype:

```
NPError NPP_DestroyStream(NPP instance, NPStream *stream, NPReason reason);
```

Processing stream data occurs in either NPP_Write() or NPP_AsFile(), depending on whether the stream in question was a NP_NORMAL/NP_ASFILE or NP_ASFILEONLY stream respectively. The mechanics of working with stream data are beyond the scope of this paper, and will not be discussed further.

NPRuntime Basics

The NPRuntime is an addition to the NPAPI that provides a uniform interface for allowing plugins to expose scriptable objects to the DOM. Prior to the introduction of the NPRuntime, there were other methods that allowed plugins to be exposed to both Java and scripting bridges – LiveConnect and XPCOM. Both of these technologies, while still supported to an extent, are considered deprecated and beyond the scope of this paper.

Plugins that wish to offer scriptable functionality do so via the use of the NPP_GetValue() function. Essentially, this function is utilized by the browser to query a plugin for a number of well-known properties. It has the following prototype:

```
NPError NPP_GetValue(NPP instance, NPPVariable variable, void *ret_value);
```

The variable parameter indicates the class of information that is to be retrieved from the plugin. Information such as the plugin's name, description, or a handle to the instance window are among the possible attributes that can be retrieved. When the NPRuntime component was introduced, a variable was added to the enumeration of possible variables that can be queried for – namely, NPPVpluginScriptableNPObject, which has a numeric value of 15. When this value is queried for, the plugin can elect to return a pointer to an NPObject that encapsulates the plugin's scripting abilities. (This object will be explored in greater detail momentarily.) This is achieved by placing a pointer to an NPObject in the ret_value parameter. When a query occurs for NPPVpluginScriptableNPObject, the ret_value parameter is actually interpreted as an NPObject **. Plugins that do not have any scriptable abilities simply return an error when NPP_GetValue() is called with the variable parameter set to NPPVpluginScriptableNPObject.

Scriptable Objects

As mentioned previously, objects are exposed through the utilization of NPObject structures, which are defined in npruntime.h as follows:

```
struct NPObject {
    NPClass *_class;
    uint32_t referenceCount;
    /*
     * Additional space may be allocated here by types of NPObjects
     */
};
```

The real functionality is accessible from the encapsulated NPClass object, also defined in npruntime.h as follows:

```
struct NPClass
{
    uint32_t structVersion;
    NPAllocateFunctionPtr allocate;
    NPDeallocateFunctionPtr deallocate;
    NPInvalidateFunctionPtr invalidate;
    NPHasMethodFunctionPtr hasMethod;
    NPInvokeFunctionPtr invoke;
    NPInvokeDefaultFunctionPtr invokeDefault;
    NPHasPropertyFunctionPtr hasProperty;
    NPGetPropertyFunctionPtr getProperty;
    NPSetPropertyFunctionPtr setProperty;
    NPRemovePropertyFunctionPtr removeProperty;
    NPEnumerationFunctionPtr enumerate;
    NPConstructFunctionPtr construct;
};
```

Each of these functions implements vital functionality for object manipulation from JavaScript; the relevant parts of this API are discussed below.

Object Initialization and Destruction

First, we will consider initialization. Typically, a scriptable object is created by defining an `NPClass` structure with all of the relevant functions implemented, and then calling the browser side function `NPN_CreateObject()`. This function has the following prototype:

```
NPObject *NPN_CreateObject(NPP npp, NPClass *aClass)
```

As can be seen, NPN_CreateObject() takes an instance pointer as its first parameter (which we will explore later), and a pointer to the NPClass structure as its second parameter. It simply creates an NPObject wrapper around the NPClass object and returns it. If the NPClass object has defined the allocate() callback, then it will be called to allocate the memory for the NPObject structure that the NPN_CreateObject() function returns. This allocation callback feature allows the developer to allocate additional space to hold any context-specific information about the object in a structure that wraps an NPObject. A standard technique is to represent an object as a C++ class, as shown:

```
// MyObject derives from NPObject –
// It will be exposed as a scriptable object

class MyObject : public NPObject
{
public:

      // Definition of the objects behaviors
      static NPClass myObjectClass =
      {
            NP_CLASS_STRUCT_VERSION,
            Allocate,
            Deallocate,
            Invalidate,
            HasMethod,
            Invoke,
            InvokeDefault,
            HasProperty,
            GetProperty,
            SetProperty,
      };

      // Call this function from NPP_GetValue() to retrieve the
      // scriptable object
      // It will create an NPObject wrapping the myObjectClass NPClass
      // It will also call Allocate() to allocate the NPObject

      static MyObject *Create(NPP npp)
      {
            MyObject *object;
            object = reinterpret_cast<MyObject *>
                  (NPN_CreateObject(npp, &myObjectClass));
      }

      // The Allocate() function creates an instance of MyObject,
      // so we can initialize any private variables for MyObject etc..
      // Note that the Allocate() function needs to be static

      static NPObject *Allocate(NPP npp, NPClass *class)
```

```
      {
            return new MyObject(npp);
      }

      .. other methods ..
};
```

The other noteworthy detail of creating objects is that the reference count member of the NPObject structure will be initialized to 1, and the member will be incremented each time the object is passed to the browser side function NPN_RetainObject(), which is defined as follows:

```
NPObject *NPN_RetainObject(NPObject *obj);
```

This function can be considered an analog of AddRef() for Microsoft COM objects.

When an object is no longer needed, the browser side function NPN_ReleaseObject() is called, which does the reciprocal operation of NPN_CreateObject(). The reference count variable is decremented, and if it reaches 0, the object will be de-allocated. If the NPClass structure pointed to in the NPObject being released contains a deallocate() callback, this will be used to destroy the object. Otherwise, the default system allocator will release the memory.

Object Behavior

The most important feature of an object is the behaviors it exposes. There are two distinct types of attributes an object may expose: properties and methods. A defined property is an attribute of an object that may be set or retrieved. It is manipulated in scripting as you would expect any other DOM-object's properties to be:

```
Plugin.property = setVal;     // set the property
retVal = Plugin.property;     // retrieve the property
delete Plugin.property;       // remove the property
```

Internally, performing any of these actions in a script will cause the invocation of two of the four defined property-related functions from the `NPClass` object that defines the object:

```
bool HasProperty(NPObject *obj, NPIdentifer name)
bool GetProperty(NPObject *obj, NPIdentifier name, NPVariant *result)
bool SetProperty(NPObject *obj, NPIdentifier name,  NPVariant *value)
bool RemoveProperty(NPObject *obj, NPIdentifier name)
```

Whether setting or retrieving a property, the first action that the browser takes is to check whether the property is supported, and this is done by calling the HasProperty() method with the name of the property that will be manipulated. The NPIdentifier data type is used to resolve the property or method and contains a hash value of the name rather than the value of the name. If the requested name isn't supported, an error is returned to the script runtime. Assuming HasProperty() succeeds, GetProperty() or SetProperty() is then called, depending on whether is being retrieved or set. In the case of retrieval, a pointer to the property's value is placed in the result parameter of GetProperty(), which will be interpreted as a return value by the script runtime (retVal in the previous scripting example). Conversely, when a property is being set, the value parameter will be interpreted as the value that the property is being set to (setVal in the previous scripting example). Lastly, a property can be removed using the delete syntax noted above. In practice, this functionality is rarely implemented. Note that the obj parameter passed as the first argument to all of these functions is a pointer to the object itself. The process of setting and getting properties is depicted in Figure 19.

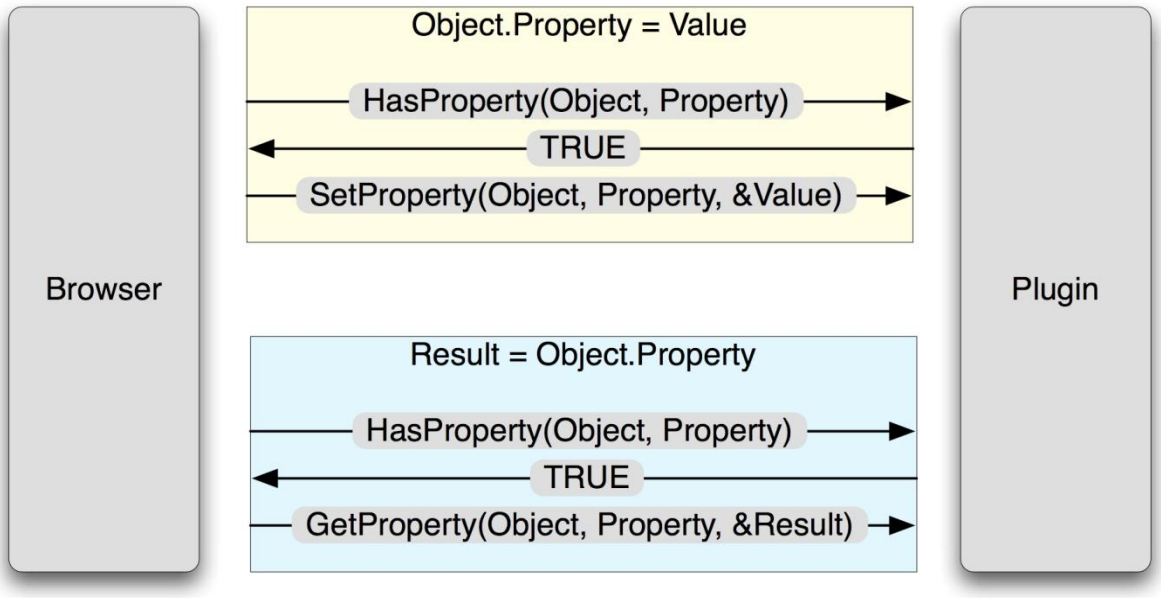


Figure 19: Getting and Setting Properties of an NPObject

Method invocations are implemented in a similar fashion to property manipulation, using three methods defined in the `NPClass` structure:

```
bool HasMethod(NPObject *obj, NPIdentifier name)
bool Invoke(NPObject *obj, NPIdentifier name, const NPVariant *args, uint32_t
      argCount, NPVariant *result)
bool InvokeDefault(NPObject *obj, const NPVariant *args, uint32_t argCount,
      NPVariant *result)
```

As with properties, when a method is called, the browser first calls HasMethod() to see if the plugin has the given method defined. Assuming this call is successful, the Invoke() function is then called. The Invoke() function takes the method name that is being called in the name parameter, followed by an array of arguments, followed by a count indicating the size of the argument array, and finally a pointer to a variant that will contain a result of the invocation. The InvokeDefault() function is used when the plugin object is executed as if it were a method, like in the following JavaScript code snippet:

```
var pluginobj = document.getElementById("plugin");
var result = pluginobj(args);
```

Parameter Passing

As we saw in the previous section, an object may define behavior in terms of properties and methods that are available to the scripting host. In both cases, parameters are passed to and from the NPAPI entry points as NPVariants. NPVariants are basically an opaque data structure used to represent different variables that can be readily imported or exported from scripting engines such as JavaScript. The NPVariant structure is defined as follows:

```
typedef struct _NPVariant {
    NPVariantType type;
    union {
        bool boolValue;
        int32_t intValue;
        double doubleValue;
        NPString stringValue;
        NPObject *objectValue;
    } value;
} NPVariant;
```

As can be seen, the structure is a very simple type/union structure, much like the VARIANT data structure that is pervasive on Microsoft Windows platforms. All of the data types here are either basic types or NPObjects (previously discussed), with one exception - the NPString value, which is defined as follows:

```
typedef char NPUTF8;
typedef struct _NPString {
    const NPUTF8 *utf8characters;
    uint32_t utf8length;
} NPString;
```

The value contained within the union is defined by the NPVariant's type, which is defined as one of the following:

```
typedef enum {
    NPVariantType_Void,
    NPVariantType_Null,
    NPVariantType_Bool,
    NPVariantType_Int32,
    NPVariantType_Double,
    NPVariantType_String,
    NPVariantType_Object
} NPVariantType;
```

The NPAPI provides a number of standardized macros for manipulating NPVariant data structures. These macros, defined in npruntime.h, are divided into three categories. The first category is for testing the type of an NPVariant, and are of the form: NPVARIANT_IS_XXX(), where XXX is the object type to check:

```
#define NPVARIANT_IS_VOID(_v)    ((_v).type == NPVariantType_Void)
#define NPVARIANT_IS_NULL(_v)    ((_v).type == NPVariantType_Null)
#define NPVARIANT_IS_BOOLEAN(_v) ((_v).type == NPVariantType_Bool)
#define NPVARIANT_IS_INT32(_v)   ((_v).type == NPVariantType_Int32)
#define NPVARIANT_IS_DOUBLE(_v)  ((_v).type == NPVariantType_Double)
#define NPVARIANT_IS_STRING(_v)  ((_v).type == NPVariantType_String)
#define NPVARIANT_IS_OBJECT(_v)  ((_v).type == NPVariantType_Object)
```

For example, testing if a particular variant is a string could be achieved using the NPVARIANT_IS_STRING() macro. The second category is for extracting the value from an NPVariant, and the macro names are of the form NPVARIANT_TO_XXX():

```
#define NPVARIANT_TO_BOOLEAN(_v) ((_v).value.boolValue)
#define NPVARIANT_TO_INT32(_v)   ((_v).value.intValue)
#define NPVARIANT_TO_DOUBLE(_v)  ((_v).value.doubleValue)
#define NPVARIANT_TO_STRING(_v)  ((_v).value.stringValue)
#define NPVARIANT_TO_OBJECT(_v)  ((_v).value.objectValue)
```

Lastly, there are macros used to store data into an NPVariant variable. These macros are of the form XXX_TO_NPVARIANT(). This last category is primarily used to fill out the result NPVariant for the GetProperty(), Invoke(), and InvokeDefault() functions.

Marshalling and Type Resolution

So, how do scripting hosts pass data to and from plugins? The answer is that a marshalling layer is required to interpret objects from the scripting host and convert them to types that the plugin understands, and vice versa. Obviously, this layer is implementation dependant, and varies from browser to browser. This section will give a brief overview of the Mozilla Firefox marshalling layer that facilitates communication between JavaScript and scriptable objects.

The conversions required for NPRuntime plugins are actually quite simple, as NPObject types map precisely to those supported by JavaScript natively in most cases. The marshalling is all contained within a single file in the Firefox source tree: mozilla/modules/plugin/base/src/nsJSNPRuntime.cpp. To achieve the two primary objectives of converting JavaScript variables to NPVariants and vice versa, an object proxying method is employed, which is discussed below.

When a property is being set or a method is being invoked, the JavaScript objects being passed to the plugin need to be converted to NPVariants. In the case of the basic types, this conversion is a straightforward procedure of transplanting a literal value from JavaScript into an NPVariant structure.

```
if (JSVAL_IS_PRIMITIVE(val)) {
    if (val == JSVAL_VOID) {
      VOID_TO_NPVARIANT(*variant);
    } else if (JSVAL_IS_NULL(val)) {
      NULL_TO_NPVARIANT(*variant);
    } else if (JSVAL_IS_BOOLEAN(val)) {
      BOOLEAN_TO_NPVARIANT(JSVAL_TO_BOOLEAN(val), *variant);
    } else if (JSVAL_IS_INT(val)) {
      INT32_TO_NPVARIANT(JSVAL_TO_INT(val), *variant);
    } else if (JSVAL_IS_DOUBLE(val)) {
      DOUBLE_TO_NPVARIANT(*JSVAL_TO_DOUBLE(val), *variant);
```

The code to handle this conversion is located in JSValToNPVariant(). In the case of strings, a little extra work is done to handle UTF-8 conversions.

```
    } else if (JSVAL_IS_STRING(val)) {
      JSString *jsstr = JSVAL_TO_STRING(val);
      nsDependentString str((PRUnichar *)::JS_GetStringChars(jsstr),
                            ::JS_GetStringLength(jsstr));

      PRUint32 len;
      char *p = ToNewUTF8String(str, &len);

      if (!p) {
        return false;
      }

      STRINGN_TO_NPVARIANT(p, len, *variant);
```

Finally, there are JavaScript objects. When these are being passed as arguments, an NPObject structure is created that wraps the JavaScript object. The functionality of the wrapper object is defined in the NPClass structure sJSObjWrapperClass, which contains methods that proxy requests over to the JavaScript engine. For example, if NPP_GetProperty() is called on the wrapper object, it will retrieve the instance of the JavaScript object being wrapped, and allows the JavaScript engine to internally handle the specifics. This process is illustrated in the Figure 20.

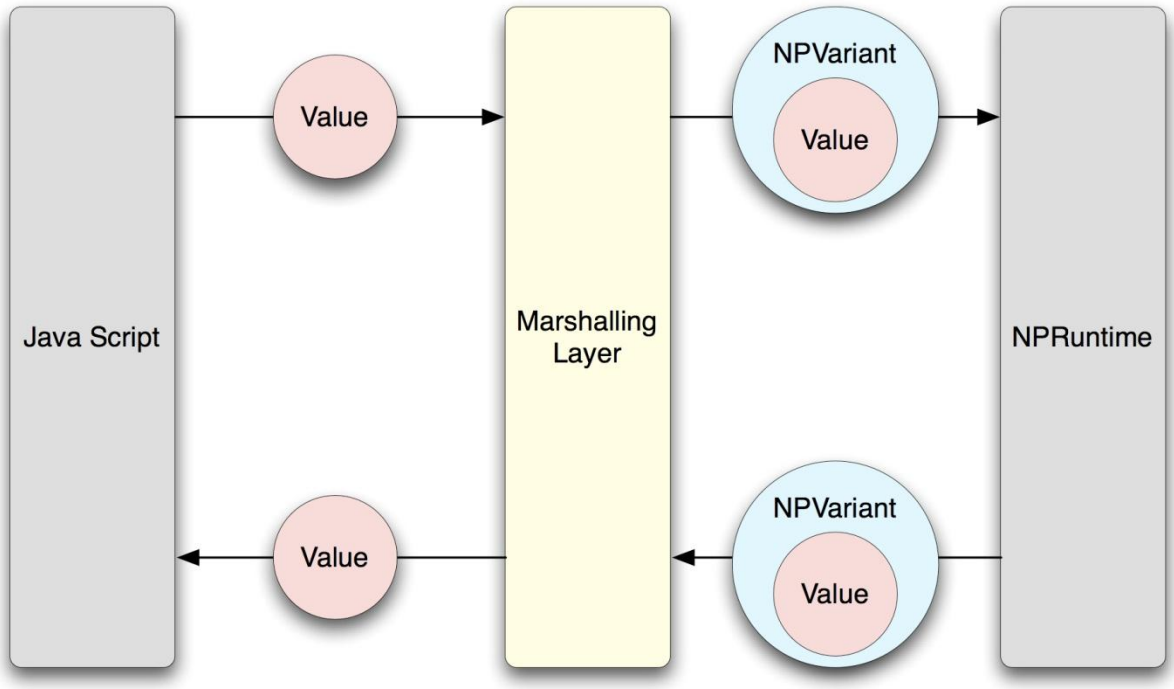


Figure 20: JavaScript Objects Encapsulated by the Marshalling Layer before being passed to the NPRuntime

Similarly, when objects are being converted from NPVariants back into JavaScript, the NPVariantToJSVal() function will copy immediate values back into JavaScript objects, or create a JavaScript object that proxies calls over to the functionality exposed by the NPObject. This proxy class is implemented using the sNPObjectJSWrapperClass structure.

Attack Surface

In the context of the NPAPI, the attack surface can be broken down into roughly three key areas, which are:

- Standard Plugin entry points
- Entry points to exposed scriptable objects, and
- Marshalling layers within the browser itself

The standard plugin entry points can be summarized as those exposed by the "Netscape Plugin" structure (ie the NPP_* functions). We have already examined a relatively large attack surface for standard entry points, particularly supplying parameters to a plugin instance via NPP_New(), or data retrieved in URL streams (which are mostly processed by the NPP_Write() function when NP_NORMAL streams are received, and NPP_StreamAsFile() when NP_ASFILEONLY streams are received). Although this is a somewhat large attack surface, it is also the explicit attack surface that most security research has focused on up until this point. As such, this paper will not deal much with these entry points except where they provide some context to the interoperability attacks that will be discussed.

The entry points exposed by scriptable objects are perhaps the most expansive attack surface that has had little treatment in the past. Functions that implement script interaction for the object are going to be the most obvious attack vectors, such as Invoke(), InvokeDefault(), GetProperty(), and SetProperty() for a given NPObject. We must also consider in this attack surface the less obvious entry points for interoperability – primarily places where a plugin accesses the DOM hierarchy using the NPN_GetProperty() function.

Lastly, each browser that implements an NPAPI runtime must provide some marshalling layer for converting objects from scripting language runtimes to NPVariants and vice versa. This binding layer also presents a plethora of opportunity for attackers to target for the purpose of vulnerability discovery.

Section III: Attacks on Interoperability

Architectures designed to permit interoperability are difficult to implement. There is a new layer of problems that must be avoided when developing code and, accordingly, this situation gives rise to new opportunities for attackers to undermine system security. The following subsections will enumerate vulnerability classes that specifically arise when managing data over interoperability layers. The classes being discussed are:

1. Object Retention vulnerabilities
2. Type Confusion vulnerabilities, and
3. Transitive Trust vulnerabilities.

The research presented in both the Type Confusion and Transitive Trust classes significantly extend any previous use of these terms – to an extent that warrants viewing them as new vulnerability classes. Standard types of vulnerabilities such as integer width problems and buffer overflows are of course also present in the context of data marshalling, but will not be discussed in this paper, since they are well understood and a large body of literature already exists that sufficiently illuminate those topics.

Interoperability Attacks I: Object Retention Vulnerabilities

Data being communicated between cohesive modules can either be simple literal values (such as integers or Booleans), or complex data structures (such as COM objects). For the latter case, the runtime must have a method for managing the lifespan of an object. The general strategy of managing an object's lifespan is to use a reference counting primitive. Such a strategy places responsibility on consumers of the object to signal when they need it and when they are finished with it. It follows then that if a consumer fails to correctly report object usage, there are potential opportunities for memory management vulnerabilities. Generally speaking, mismanagement of an objects lifespan happens in two scenarios:

1. Not retaining a reference to an object when it is required, thus risking memory being de-allocated too early, and
2. Not releasing a reference to an object when it is required, resulting in memory leaks and also potentially exploitable scenarios (discussed shortly).

This section will describe how both of these code constructs typically manifest in two different real-world plugin architectures. The reader should note that the vulnerable code constructs in this section are likely to be found in both plugin objects and marshalling layers themselves, as these interfaces often need to retain references to objects and generate new objects while performing conversions during the coercion process.

Microsoft Object Retention Vulnerabilities

The Microsoft plugin architecture makes extensive use of COM objects and VARIANTs to define and pass objects between the various components within the browser. Indeed, JavaScript obects are represented natively in the language runtime as COM objects, whereas VBScript objects are represented as VARIANTs. A method or property exposed by an ActiveX object is accessed by calling the IDispatch::Invoke() method of the object, which receives parameters to the destination function as an array of VARIANTs. (Note that with ActiveX controls, properties are actually exposed as a pair of method calls that have names of the form get_XXX() and put_XXX(), where XXX is the name of the property. These two functions retrieve and set the property respectively.) Objects contained within the VARIANTs can really be any type and value, but most commonly they are either primitive types (such as integers or strings), or COM interfaces that represent complex objects. Since JavaScript represents objects internally as IDispatch (or more accurately, IDispatchEx) COM interfaces, VT_DISPATCH VARIANTs will be the most common COM-based VARIANTs passed to typical controls, in the context of a browser.

COM objects maintain an internal reference count, and it is manipulated externally by the IUnknown::AddRef() and IUnknown::Release() methods, which increment and decrement the reference count respectively. Once the reference count reaches 0, the object will delete itself from memory. Object retention errors in ActiveX controls are a result of mis-management of the object's reference count. This section describes the typical mistakes made by developers when dealing with objects whose lifespan is greater than the scope of a single function call.

ActiveX Object Retention Attacks I: No Retention

The most obvious mistake a control can make with regard to object retention is to neglect to add to the reference count of a COM object that it intends to retain. When an ActiveX function takes a COM object as a parameter, the marshalling layer has already called IUnknown::AddRef() on the received object to ensure that it won't be deleted by competing threads. However, the marshaller will also release the interface after the plugin function has returned. Therefore, a plugin object wishing to retain an instance of a COM object beyond the scope of a method must call the IUnknown::AddRef() function before the method returns. Calling IUnknown::QueryInterface() is also sufficient, as this function will (or at least, should) call IUnknown::AddRef() for the object as well. Failure to call either of these functions can result in potential stale pointer vulnerabilities. The code below shows an example of such a problem.

```
HRESULT CMyObject::put_MyProperty(IDispatch *pCallback)
{
      m_pCallback = pCallback;
      return S_OK;
}

HRESULT CMyObject::get_MyProperty(IDispatch **out)
{
      if(out == NULL || *out == NULL || m_pCallback == NULL)
            return E_INVALIDARG;

      *out = m_pCallback;
      return S_OK;
}
```

The put_MyProperty() function in this code stores an IDispatch pointer which can later be retrieved by the client application using the get_MyProperty() function. However, since AddRef() is never used, there is no guarantee that the pCallback function will still exist when the property is read back by the client. If every other reference to the object is removed, the object will be de-allocated, leaving m_pCallback pointing to stale memory.

VARIANT Shallow Copies

When a VARIANT object is duplicated, it is typically done with VariantCopy(), but just a simple memcpy() is also used in many cases. VariantCopy() is the preferred method, since it will do an object-aware copy – if the VARIANT being copied is a string, it will duplicate the memory. If the object being copied is an object, it will add a reference count. In contrast, memcpy() obviously performs a shallow copy – if the VARIANT contains any sort of complex object, such as an IDispatch, a pointer to the object will be duplicated and utilized without adding an additional reference to the object. If the result of this duplicated VARIANT is retained, the object being pointed to could be deleted, if every other instance of that object is released. The following code demonstrates this vulnerable construct.

```
HRESULT CMyObject::put_MyProperty(VARIANT src)
{
      HRESULT hr;

      memcpy((void *)&m_MyProperty, (void *)&src, sizeof(VARIANT));
```

```
	return S_OK;
}

HRESULT CMyObject::get_MyProperty(VARIANT *out)
{
	HRESULT hr;

	if(out == NULL)
		return E_FAIL;

	VariantInit(out);

	memcpy(out, (void *)&m_MyProperty, sizeof(VARIANT));

	return S_OK;
}
```

There is also a more subtle variation on the attack – this time using VariantCopy(). In some ways, VariantCopy() can be also be considered a shallow copy operation, in that any VARIANT that has the VT_BYREF modifier will not be deep-copied; just the pointer will be copied. Consider the following code:

```
HRESULT CMyObject::put_MyProperty(VARIANT src)
{
	HRESULT hr;

	VariantInit(&m_MyProperty);

	hr = VariantCopy(&m_MyProperty, &src);

	if(FAILED(hr))
		return hr;

	return S_OK;
}

HRESULT CMyObject::get_MyProperty(VARIANT *out)
{
	HRESULT hr;

	if(out == NULL)
		return E_FAIL;

	VariantInit(out);

	hr = VariantCopy(out, &m_MyProperty);

	if(FAILED(hr))
```

```
            return hr;

      return S_OK;
}
```

This example shows a sample ActiveX property that simply takes a VARIANT and stores it, and optionally returns it to the user. The problem with this code is that VariantCopy() is used rather than VariantCopyInd(). If a VARIANT is supplied that has the type (VT_BYREF|VT_DISPATCH) for example, a simple pointer copy is performed. If the VT_DISPATCH object being pointed to is subsequently deleted, then you are left with a VARIANT pointing to an IDispatch object that no longer exists. If an attempt to get this property is subsequently made, the user will retrieve a VARIANT with a stale pointer, leading to the possibility of memory corruption.

The ActiveX Marshaller

In order to know the exact semantics of what happens to an object when it is passed as a parameter to an ActiveX control, you need to pay careful attention to what types the target function is expecting. When an ActiveX function expects a VARIANT as a parameter, the marshalling code does not do any sort of deep copy - it uses neither VariantCopy() nor VariantCopyInd(). So, receiving VARIANTs can be particularly dangerous if they contain COM interfaces that are operated upon beyond the method's scope. Furthermore, if an ActiveX function allows an indirect pointer to a COM object as a parameter - that is, (VT_BYREF|VT_DISPATCH) or equivalent, the object being referenced will have its reference count incremented by the marshaller (and released upon function returned). So if a VARIANT value is passed to an ActiveX control of type (VT_BYREF|VT_DISPATCH), it will not have its reference count incremented if the function takes a VARIANT, but it will have its reference count incremented if the function takes a IDispatch ** (or even an IDispatch *). This algorithm is somewhat counterintuitive, which increases the likelihood that mistakes will occur as a result.

ActiveX Object Retention Attacks II: Release Failure

Failure to release an object essentially amounts to a memory leak. These failures occur when a COM interface is referenced through IUnknown::AddRef() or IUnknown::QueryInterface(), and are later discarded without calling the corresponding IUnknown::Release() function. Triggering code paths that operate this way can allow an attacker to consume arbitrary amounts of memory, but more usefully increment the reference count of an object an unlimited number of times. On a 32-bit machine, by executing the vulnerable code path 0xFFFFFFFF times, an integer overflow can be triggered in the object's reference count. Following that, any call to IUnknown::Release() will cause the object to be de-allocated, which, again, can lead to stale pointer problems. The following code is based on an example we previously used; however, it has been modified to demonstrate problems with failing to release an object.

```
HRESULT CMyObject::put_MyProperty(IDispatch *pCallback)
{
      if(pCallback == NULL)
            return E_INVALIDARG;

      pCallback->AddRef();
      m_pCallback = pCallback;
      return S_OK;
}

HRESULT CMyObject::get_MyProperty(IDispatch **out)
{
      if(out == NULL || *out == NULL || m_pCallback == NULL)
            return E_INVALIDARG;

      *out = m_pCallback;
      return S_OK;
}
```

This example correctly adds a reference to the new callback object when it is set. However, the previous value held in m_pCallback, if one existed, is overwritten without being released. Therefore, an attacker can set this property a large number of times and eventually trigger an integer overflow in the reference count variable. Let's try fixing it in the following example:

```
HRESULT CMyObject::put_MyProperty(IDispatch *pCallback)
{
	if(pCallback == NULL)
		return E_INVALIDARG;

	pCallback->AddRef();

	if(m_pCallback != NULL)
		m_pCallback->Release();

	m_pCallback = pCallback;
	return S_OK;
}

HRESULT CMyObject::get_MyProperty(IDispatch **out)
{
	if(out == NULL || *out == NULL || m_pCallback == NULL)
		return E_INVALIDARG;

	*out = m_pCallback;
	return S_OK;
}
```

The above example adds a Release() call to correctly release any previously held objects, and so no memory leak occurs. Astute readers will notice that this code actually still has a stale pointer problem. The get_MyProperty() function doesn't add a reference to the interface being distributed back to the scripting engine. This can be problematic if the only reference to that interface is held by the plugin, and the plugin releases it. Consider the following JavaScript snippet:

```
axObject.MyProperty = new Object();
var x = axObject.MyProperty();
axObject.MyProperty = new Object();
```

This JavaScript code results in the following actions taking place:

1. put_MyProperty retains the only reference to the object we created.
2. The 'x' variable receives the IDispatch pointer, still there is only one copy of it
3. Setting MyProperty will cause the old object to be deleted, even though 'x' still points to it!

Mozilla Object Retention Vulnerabilities

The NPAPI has a more simplistic model for object marshalling than the COM architecture. As described in the technology overview section of this paper, JavaScript objects cannot be passed to a plugin directly, but rather are wrapped in an object format that is understood by the NPAPI – the NPObject. Recall the NPObject structure has a reference count, which is manipulated with NPN_RetainObject(), and NPN_ReleaseObject(). Object retention vulnerabilities in NPAPI-based browsers stem from the misuse of these two functions, and are described below.

NPAPI Object Retention Attacks I: No Retention

Like ActiveX controls, NPAPI modules need to maintain references to objects received as input parameters whenever those objects will be stored for an extended period of time. As mentioned in the technology overview, NPObjects are created by the marshalling layer to wrap JavaScript objects. If a particular JavaScript object has been wrapped by an NPObject in the past, that same NPObject will be reused. Furthermore, NPObjects can be created by the plugin using NPN_CreateObject(), which might then be passed back to the user at some point. In either case, if a plugin needs to maintain a pointer to an object, they are required to call NPN_RetainObject(), passing a pointer to the NPObject in question as the parameter. Failure to do so results in a potential stale pointer vulnerability in the plugin. The following code is an example of an object retention vulnerability using the NPAPI API.

```
bool SetProperty(NPObject *obj, NPIdentifier name, const NPVariant *variant)
{
      if(name == kTestIdent)
      {
            if(!NPVARIANT_IS_OBJECT(*variant))
                  return false;

            gTestObject = NPVARIANT_TO_OBJECT(*variant);
            return true;
      }
      return false;
}

bool GetProperty(NPObject *obj, NPIdentifier name, NPVariant *result)
{
      VOID_TO_NPVARIANT(*result)

      if(name == kTestIdent)
      {
            if(!NPVARIANT_IS_OBJECT(*result))
                  return false;

            if(gTestObject == NULL)
                  NULL_TO_NPVARIANT(*result);
            else
                  OBJECT_TO_NPVARIANT(*result, gTestObject);
```

```
            return true;
      }
      return false;
}
```

As can be seen, the SetProperty() method retains a pointer to an object but fails to call NPN_RetainObject(). A malicious user could exploit this problem by executing the following steps:

1. Create an object of some kind
2. Set the vulnerable property using that object
3. Delete the object
4. Get the vulnerable property

NPAPI Object Retention Attacks II: Release Failure

Like with ActiveX controls, release failure problems can occur in the NPAPI as well. It occurs when an object is retained using NPN_RetainObject() but never released using NPN_ReleaseObject(). Again, by triggering this code path many times, an opportunity to overflow the reference counter will be available, potentially leading to stale pointer problems. The following code is a slightly modified version of the previous example which demonstrates the problem.

```
bool SetProperty(NPObject *obj, NPIdentifier name, const NPVariant *variant)
{
      if(name == kTestIdent)
      {
            if(!NPVARIANT_IS_OBJECT(*variant))
                  return false;

            gTestObject = NPN_RetainObject(NPVARIANT_TO_OBJECT(*variant));
            return true;
      }
      return false;
}
```

In the above code, NPN_RetainObject() is correctly called on the object being retrieved from the user. However, notice that gTestObject is never checked to see if it has been set previously. Any NPObject that was stored here previously is not released, and so the code contains a reference count leak. An attacker could exploit this opportunity using the following steps:

1. Create an NPObject, either by wrapping one particular JavaScript object or by using another NPObject created by the plugin
2. Create a second reference to the same object by assigning it to more than one variable in JavaScript (let's call them objX and objY).
3. Call SetProperty() 0xFFFFFFFF times to take the reference count of the NPObject from 2 to 1 (due to the integer overflow)
4. Delete one of the variables, say, objX. This will take the reference count to 0 and destroy the NPObject.
5. objY will now contain a stale NPObject reference.

The concrete examples presented concerning reference counting are browser and platform specific. However, these types of problems are symptomatic of the complexity of interoperability. In general, interoperability architectures that allow passing values by reference and allow maintaining those references are going to encounter this problem quite often. Thus, object retention is a fertile target for attackers looking to find vulnerabilities in applications that offer interoperability.

Interoperability Attacks II: Type Confusion Vulnerabilities

Type confusion vulnerabilities are, as the name implies, vulnerabilities that occur when one data type is mistaken for another. They are most often the result of mismanagement of union data types, but can also stem from type wildcards, and result in an attacker being able to either read sensitive data from a target application (ie. an information leak), or achieve unintended execution. Type confusion vulnerabilities have a higher likelihood of appearing in software components responsible for decoding complex objects of arbitrary types represented in a language agnostic format. The reason for this higher likelihood is that, when the intended effect of the code is to convert between contrived and fundamental types, the compiler's error checking is rendered impotent. Some situations where the vulnerability class will be prevalent include:

- De-serializing objects from persistent storage (such as a file)
- De-serializing objects from a networked application (such as ASN.1 encoded objects), and
- Language bindings layers charged with marshaling data between two languages that differ in their native representation

This section introduces type confusion vulnerabilities, how they occur, and the implications they have for an application's security. Auditing to locate such vulnerabilities will also be discussed, using a number of prevalent APIs as case studies, as well as real world examples of vulnerabilities uncovered by the authors.

The Basics: Type Wildcards

Fundamentally, a type confusion vulnerability results from a piece of code that performs operations on a storage area under mistaken assumptions regarding the storage area's type . Take for example, the following code:

```
int ReadFromConnection(int sock)
{
    unsigned char *Data;
    int total_size;
    int msg_size;

    total_size = 1024;
    Data = (unsigned char *)malloc(total_size);

    msg_size = recv(sock, &Data, total_size, 0);
    return(1);
}
```

The recv function expects to be able to write up to total_size bytes into the memory area specified by Data. However, in this example, the code has mistaken the parameter's type - it is passing a pointer to a pointer to an area of memory that can hold up to total_size bytes. On a 32-bit machine, the memory

area will only be able to hold four bytes of data, leading to a stack overflow. The compiler will allow this error to occur because the recv function specifies that argument two should be a void * type, which specifies that the function will accept a pointer to any type of memory, including a pointer to a pointer.

An example of precisely this type of vulnerability was discovered by one of the authors (Ryan Smith) in a Microsoft-internal version of the ATL. The problematic code is triggered when reading VARIANTs of type VT_ARRAY | VT_UI1 from a persistent stream. The following code is a rough representation of the vulnerable function.

```
inline HRESULT CComVariant::ReadFromStream(IStream *pStream)
{
	…
	hr = pStream->Read(&vtRead, sizeof(VARTYPE), NULL);
	…
	switch(vtRead)
	{
		case VT_ARRAY|VT_UI1:
			SAFEARRAYBOUND rgsaInBounds;
			SAFEARRAYBOUND rgsaBounds;
			SAFEARRAY *saBytes;
			void *pvData;

			hr=pStream->Read(&saInBounds, sizeof(saInBounds), NULL);
			if(hr<0||hr==1)
				return(hr);

			rgsaBounds.cElements = rgsaInBounds.cElements;
			rgsaBounds.lLbound = 0;
			saBytes = SafeArrayCreate(VT_UI1, 1, rgsaBounds);
			if(saBytes == NULL)
				return(E_OUTOFMEMORY);

			hr = SafeArrayAccessData(saBytes, &pvData);
			if(hr < ERROR_SUCCESS)
			    return(hr);

			hr=pStream->Read(&pvData, rgsaInBounds.cElements, NULL);
			...
	}
}
```

The code above reads data from an IStream, incorrectly passing a pointer to a pointer to the destination buffer, rather than a pointer to the destination buffer (that is, it passes &pvData as the buffer parameter instead of pvData). On a 32-bit system, if the amount of data read is larger than 4 bytes, then stack corruption will occur. This process is visually depicted in Figure 21. Given that this code has been around for a long time and has been distributed across a large number of COM components, it is evident that type confusion bugs such as the previous example are afforded little attention, and are quite subtle.

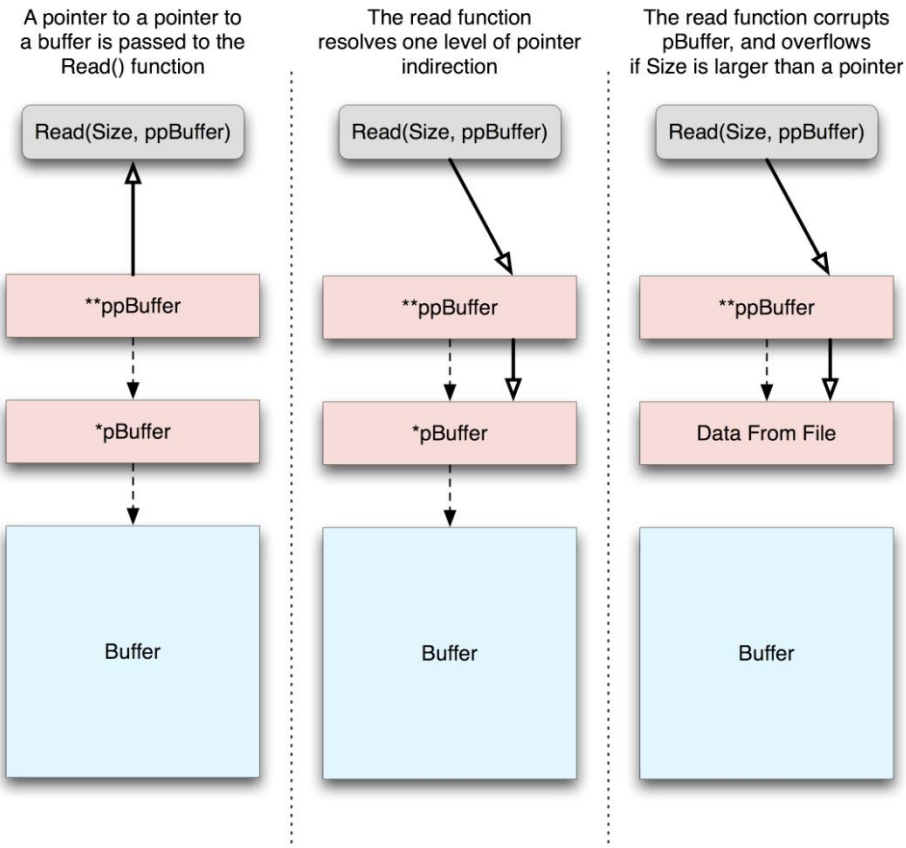


Figure 21: Diagram depicting memory corruption resulting from a type confusion vulnerability

Authors' Comment: When writing up the example code for this vulnerability, the author accidentally wrote the wrong values into the parameters for pStream->Read() - another type confusion bug! When found in peer review, another author corrected it, putting in different, but equally wrong values! I guess this code was never meant to be safe.

The Basics: Union Constructs

As alluded to in the introduction, the bulk of type confusion vulnerabilities primarily occur due to the misuse of union data types. In C and C++, a union data type is similar to a struct data type – it consists of a number of members of differing names and types, each of which can be referred to individually. However, unlike the struct type, union members all occupy the same location in memory, thus making their usage mutually exclusive. The existence of these types therefore introduces the possibility of mistakenly referring to a member of a union that is invalid, such as in the following example.

```
struct VariantType
{
      union {
            char *string_member;
            int int_member;
```

```
      };
};

int Respond(int clientSock, struct VariantType *pVar);
int HandleNetworkMsg(int clientSock);


int Respond(int clientSock, struct VariantType *pVar)
{
    int len;
    int sentLen;

    if(pVar == NULL)
        return(0);
    len = strlen(pVar->string_member);
    sentLen = send(clientSock, pVar->string_member, len+1, 0);
    if(sentLen != len+1)
        return(0);
    return(1);
}

int HandleNetworkMsg(int clientSock)
{
    struct VariantType myData;
    char inBuf[1024];
    int msgSize;
    int respCode;

    memset(inBuf, 0x00, sizeof(inBuf));
    msgSize = recv(clientSock, inBuf, sizeof(inBuf), 0);
    if(msgSize < sizeof(int))
        return(0);
    memcpy(&myData.int_member, inBuf, sizeof(int));
    respCode = Respond(clientSock, &myData);
    return(respCode);
}
```

As can be seen here, an integer is stored in the union – namely, `int_member`. Subsequently, the `string_member` variable is accessed, which is of type `char *`. Clearly, treating an integer as a string is invalid. This code construct will result in the integer stored in `int_member` being incorrectly interpreted as a `char *`, thus causing the application to act on an arbitrary part of memory of the attackers choosing as if it were a string. The compiler allows this code to compile without warning because the union type is meant to facilitate access to a section of memory using different fundamental data types, and puts the impetus on the programmer to keep track of which union member is appropriate to access at any given point.

Of course, a code construct as seen in the above example would occur quite infrequently in shipping code. But, when a union is designated a value, how do consumers of the union know what data kind of data is contained within the union? The answer is they don't; there are no intrinsic language facilities to determine this information. Instead, the programmer must extrinsically indicate what type of data is contained within the union. The programmer typically accomplishes this task by utilizing data structures such as the one below.

```
struct VariantType
{
    unsigned long TypeValueBits;
    union {
        char *str_member;
        int *pint_member;
        class *class_member;
        unsigned long ulong_member;
    };
};
```

This structure has a type member, TypeValueBits, which indicates the type of data that is contained within the union. Indeed, the VARIANT data type pervasive throughout Windows is exactly this format, and shall be revisited later. The essence of a type confusion vulnerability is to either desynchronize the member that indicates which union member is appropriate to access with what is contained inside the union, or locating code where the type field is incorrectly interpreted.

Microsoft Type Confusion Vulnerabilities: VARIANTs

As we previously saw in the technology overview of this paper, the VARIANT data structure is used extensively throughout Microsoft code as a standardized, language agnostic method of representing a variety of data types. The API for manipulating VARIANT data structures has been introduced in the overview section of this paper. We will now explore how mismanagement of VARIANT structures either directly or through the well-defined API can lead to a number of subtle type confusion scenarios.

VARIANT Type Confusion Attacks I: Permissive Property Maps

As was discussed earlier, Microsoft's ATL helps developers rapidly develop COM components by distributing template code for a collection of interfaces. Microsoft has written the template code in an abstract manner, which allows the template code to be used in a large variety of situations; however, there are also subtle consequences of utilizing some of the available code. Specifically, the manner in which the developer used to specify COM object properties using property maps has some subtle nuances that could potentially lead to opportunities for an attacker to perform type confusion attacks.

Consider the following macros available in version 9 of the Microsoft ATL, which can be used for specifying individual properties within a property map.

```
struct ATL_PROPMAP_ENTRY
{
      LPCOLESTR szDesc;
      DISPID dispid;
      const CLSID* pclsidPropPage;
      const IID* piidDispatch;
      DWORD dwOffsetData;
      DWORD dwSizeData;
      VARTYPE vt;
};

#define PROP_DATA_ENTRY(szDesc, member, vt) \
            {OLESTR(szDesc), 0, &CLSID_NULL, NULL, \
            offsetof(_PropMapClass, member), \
            sizeof(((_PropMapClass*)0)->member), vt},

#define  PROP_ENTRY(szDesc, dispid, clsid) \
            {OLESTR(szDesc), dispid, &clsid, &__uuidof(IDispatch), \
             0, 0, VT_EMPTY},

#define PROP_ENTRY_EX(szDesc, dispid, clsid, iidDispatch) \
            {OLESTR(szDesc), dispid, &clsid, &iidDispatch, 0, 0, VT_EMPTY},

#define PROP_ENTRY_TYPE(szDesc, dispid, clsid, vt) \
            {OLESTR(szDesc), dispid, &clsid, &__uuidof(IDispatch), 0, 0, vt},

#define PROP_ENTRY_TYPE_EX(szDesc, dispid, clsid, iidDispatch, vt) \
            {OLESTR(szDesc), dispid, &clsid, &iidDispatch, 0, 0, vt},
```

It is important to note that neither PROP_ENTRY nor PROP_ENTRY_EX require a parameter to specify the VARIANT type. Recall from our previous discussion about persistence that when these functions are used, the persistence stream will contain two bytes that identify the serialized type preceding the serialized data. Once the member being described has been de-serialized, the ATL code will call the put property method of the IDispatch interface that the property map specifies in order to write the data to the COM object. In summary, utilizing these macros provides a possible opportunity to provide any type of VARIANT to the put method of the IDispatch interface without forcing coercion to a specific data type. If the developer fails to take into consideration that the put method may be supplied with an arbitrary VARIANT type, then using this type of property declaration can lead to possible type confusion problems. This type of vulnerability is more likely to be found in objects that aren't used in Internet Explorer, or in interfaces that implement IDispatch that are specified in the property map, but are not accessible from Internet Explorer.

Developers may also elect to use PROP_DATA_ENTRY() instead of PROP_ENTRY(). The PROP_DATA_ENTRY macro is unique, in that the data for that property is not filtered by an IDispatch interface. Instead, it is written directly to an offset within the class memory that holds the property data. If the variant type supplied to the macro is VT_EMPTY, then the persistence code will read up to the number of bytes available for the property within the class. The process for unpacking PROP_DATA_ENTRY properties versus PROP_ENTRY macros is illustrated in Figure 22.

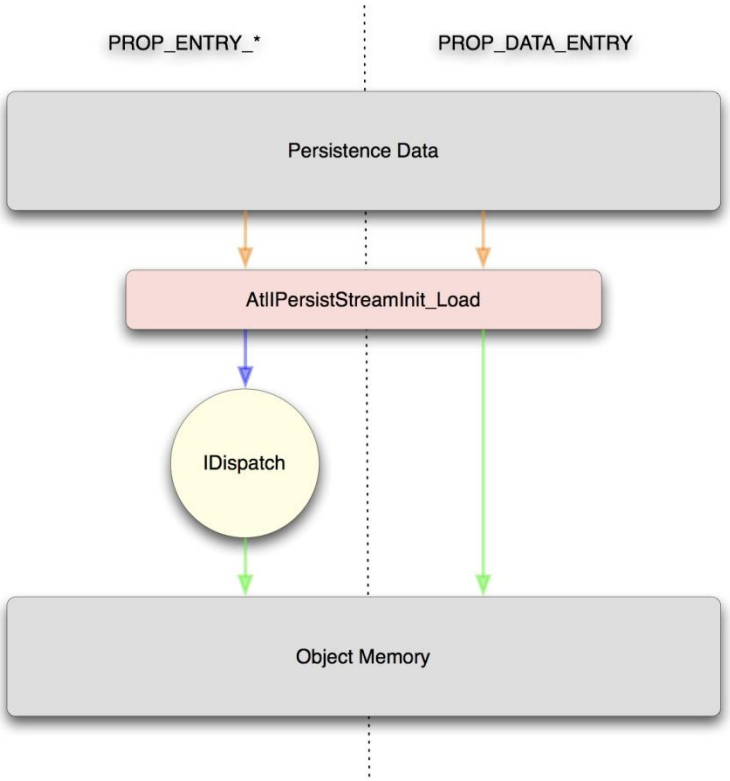


Figure 22: Diagram depicting the difference between PROP_ENTRY_* macros and PROP_DATA_ENTRY

So, the use of the PROP_DATA_ENTRY() macro provides attackers with two interesting opportunities:

1. The ability to create a property directly in the destination object's memory possibly without having any typing requirements, and
2. The ability to provide properties that have undergone absolutely no validation

If the PROP_DATA_ENTRY macro is specified in a type-less manager then these properties are quite dangerous. If they are constructed with the type specified as VT_EMPTY, then code that subsequently utilizes such properties will almost certainly contain type confusion vulnerabilities, since it has no way to validate what type of data it is operating on. For example, consider a case where a PROP_DATA_ENTRY property is intended to be a pointer to a string or some other more complex object. By specifying an integer type instead of the intended object, a type confusion vulnerability will be triggered, with the end result more than likely being arbitrary execution. Conversely, there may be a situation where a property member is expected to be an integer, but the attacker specifies a pointer instead (by specifying a string or something else). This example type confusion vulnerability will more than likely result in an information leak, and ultimately disclose the value of a pointer. These types of problems are becoming increasingly useful when attempting to bypass memory protection mechanisms found in contemporary Windows OSs.

Furthermore, it is worth considering that PROP_DATA_ENTRY properties are set directly, and hence bypass any level of validation that the put property of the IDispatch interface may enforce. This means there may be cases where setting these properties directly may circumvent the sanitization process to some degree, since it might be carried out in the put property method. Therefore, there are potential opportunities for an attacker to exploit the object in question when the property is utilized under the tenuous assumption that it is sanitized in a certain manner.

VARIANT Type Confusion Attacks II: Misinterpreting Types

One area that is prone to potential problems when dealing with VARIANT data structures is correctly interpreting the vt member. In contrast to the NPAPI variant data structures, recall that the type parameter in a VARIANT can be a basic type, or a complex type composed of bits that represent a basic type and a modifier (or two modifiers, if one of them is VT_BYREF). The misinterpretation of the vt member can occur when bit masking is performed incorrectly, leading to subtle vulnerabilities where the VARIANT's value is utilized as one type when it is in fact, another.

To illustrate this point, consider the following code:

```
#ifndef VT_TYPEMASK
#define VT_TYPEMASK 0xfff
#endif

WXDLLEXPORT bool wxConvertOleToVariant(const VARIANTARG& oleVariant,
wxVariant& variant)
{
    switch (oleVariant.vt & VT_TYPEMASK)
    {
    case VT_BSTR:
        {
            wxString str(wxConvertStringFromOle(oleVariant.bstrVal));
            variant = str;
            break;
        }

        …
```

The astute reader will notice that this code has a very obvious flaw: a type check is performed using a mask to obtain the basic type of the VARIANT. In the case of a BSTR, the string is passed to a function which basically duplicates it. The problem here is that if a modifier is used, the VARIANT will not contain a BSTR as its value parameter. If the caller of this function were to supply a VARIANT with the type (VT_BYREF|VT_BSTR) for example, it would cause a pointer to a BSTR to be placed within the VARIANT rather than a BSTR. (A BSTR is really a WCHAR * with a 32-bit length preceding it, so a BSTR * is a WCHAR **.) Therefore, utilization of any modifiers on VARIANTs passed to this function will result in a type confusion vulnerability.

Consider this slightly more subtle example:

```
SAFEARRAY *psa;
ULONG *pValue

// Test if object is an array of integers
VARTYPE baseType = pVarSrc->vt & VT_TYPEMASK;

if(  (baseType != VT_I4 && baseType != VT_UI4) ||
    ((pVarSrc->vt & VT_ARRAY) == 0) )
      return -1;

psa = pVarSrc->parray;

// operate on SAFEARRAY
SafeArrayAccessData(psa, &pValues);

...
```

This code performs some checking to ensure that an input type is an array of either signed, or unsigned integers. If it is not, then an error is signaled by returning the value -1. However, there is also a problem in this code – the check for the variant type fails to take into account that the type can have the VT_BYREF bits set. Since the VT_ARRAY modifier is not mutually exclusive with VT_BYREF, the above code has a type confusion vulnerability when dealing with a VARIANT with the type (VT_BYREF|VT_ARRAY|VT_I4). In this case, a SAFEARRAY ** will be incorrectly interpreted as a SAFEARRAY *, leading to out of bounds memory accesses.

The following code is a real-world example taken from IE (all present versions). This example is part of the core marshalling code for the DOM. The code in question is charged with verifying VARIANT parameters received from scripting hosts plugged into the DOM are correct and, if necessary, converting those parameters into the expected types. Although each DOM function takes different types of parameters, most marshalling routines, at their core, use the same function, VARIANTArgToCVar(), which takes a single VARIANT and attempts to convert it to the expected type. The vulnerable code is shown below.

```
int VARIANTARGToCVar(VARIANT *pSrcVar, int *res, VARTYPE vt, PVOID outVar,
IServiceProvider *pProvider, BOOL bAllocString)
{
      VARIANT var;

      VariantInit(&var);

      if(!(vt & VT_BYREF))
      {
            // Type mismatch - attempt conversion

            if( (pSrcVar->vt & (VT_BYREF|VT_TYPEMASK)) != vt &&
```

```
                  vt != VT_VARIANT)
            {

                  hr = VariantChangeTypeSpecial(&var, pSrcVar, vt,
                        pProvider, 0);

                  if(FAILED(hr))
                        return hr;

                  ... more stuff ...

                  return hr;
            }

            switch(vt)
            {
                  case VT_I2:
                        *(PSHORT)outVar = pSrcVar->iVal;
                        break;

                  case VT_I4:
                        *(PLONG)outVar = pSrcVar->lVal;
                        break;

                  case VT_DISPATCH:
                        *(PDISPATCH)outVar = pSrcVar->pdispVal;
                        break;

                  ... more cases ...
            }
      }
}
```

The code in question attempts to retrieve the value of an input parameter, pSrcVar, performing a type conversion if the received VARIANT isn't of the expected type given in the vt parameter. The problem in this code occurs when comparing the received input VARIANTs type with the expected type. Specifically, a test is done by comparing the expected type with the input type after the input type has been masked with (VT_BYREF|VT_TYPEMASK), or 0x4FFF. Performing this mask loses significant information, which in this case is the VT_ARRAY (0x2000) and VT_VECTOR (0x1000) modifiers. To illustrate the problem, consider the case where this function is expecting a VT_DISPATCH input type (0x0009) and the input VARIANT is an array of VT_DISPATCH types (VT_ARRAY|VT_DISPATCH, or 0x2009). Since (0x2009 & 0x4FFF) produces the result 0x0009, or VT_DISPATCH, this code will incorrectly assume it received an IDispatch object rather than an array of IDispatch objects. The result? This function signals success and returns a pointer to a SAFEARRAY which it has incorrectly evaluated as a pointer to an IDispatch interface. Thus, this code culminates to a type confusion vulnerability.

An assessor auditing for vulnerabilities in the use of VARIANT type masks must pay close attention to how the vt member of the VARIANT is manipulated. Specifically, the masking of input VARIANT types

needs to be performed with caution to ensure that information is not overlooked when performing any validation steps.

VARIANT Type Confusion Attacks III: Direct Type Manipulation

Another construct that can result in type confusion vulnerabilities is directly manipulating the vt member of a VARIANT, rather than using the API functions. Although this should be a fairly straightforward task in theory, subtle vulnerabilities can be introduced by either not correctly enforcing data types, or not correctly ensuring that a type conversion was successful. For example, the following code has been taken from Microsoft's internal version of the ATL. This code is invoked when performing de-serialization of a COM object from a persistence stream. Note that in this particular example, the VARIANT data structure is wrapped in a C++ object, CComVariant. The class member vt in this code corresponds to the vt type variable in a VARIANT structure.

The example listed above is contrived; however, the authors of this paper have identified a real-world scenario where this type of bug has occurred. Microsoft's internal version of the ATL has special code to process variants in a persistence stream that is similar to the following example.

```
inline HRESULT CComVariant::ReadFromStream(IStream* pStream)
{
	ATLASSERT(pStream != NULL);
	HRESULT hr;
	hr = VariantClear(this);
	if (FAILED(hr))
		return hr;
	VARTYPE vtRead;
	hr = pStream->Read(&vtRead, sizeof(VARTYPE), NULL);
	if (hr == S_FALSE)
		hr = E_FAIL;
	if (FAILED(hr))
		return hr;

	vt = vtRead;

    //Attempts to read fixed width data types here

	CComBSTR bstrRead;

	hr = bstrRead.ReadFromStream(pStream);
	if (FAILED(hr))
		return hr;
	vt = VT_BSTR;
	bstrVal = bstrRead.Detach();
	if (vtRead != VT_BSTR)
	{
		hr = ChangeType(vtRead);
		vt = vtRead;
	}
	return hr;
}
```

The issue with the above code is that the return value of the ChangeType() function is not checked before manually setting the variant type. This mistake allows an attacker to make the program believe a BSTR value the attacker supplied is any type not handled in the fixed width data types handler. In one scenario, an attacker can specify that a string that he has supplied should be treated as an array of VT_DISPATCH objects. When this function returns an error, the caller will attempt to free the string using the VariantClear () function. This ends up causing the program to treat the attacker supplied string as an array of vtables, a clear type confusion error, ultimately allowing for arbitrary code execution.

VARIANT Type Confusion Attacks IV: Initialization Errors

Despite being a relatively simple data structure to manipulate, VARIANTs lend themselves to misuse in certain scenarios due to the deceptive nature of parts of the API. One of the key mistakes the authors uncovered when researching VARIANT usage for this paper is the mismatching of VarintInit() and VariantClear() calls. As we mentioned earlier in the paper, the VariantInit() function is used to initialize a VARIANT structure by setting the vt member to VT_EMPTY. Conversely, VariantClear() will free the data associated with a VARIANT, taking into account what type of data is being stored there. It will subsequently set the type value of the VARIANTs to VT_EMPTY.

The important thing to notice here is that any code path that exists where VariantClear() is called on a VARIANT that has not been initialized correctly can lead to potential security problems. Why? Because VariantClear() will read the uninitialized vt member of the VARIANT and use that to decide how to operate on the uninitialized VARIANT value. For example, if the vt member was VT_DISPATCH (0x0009), VariantClear() would take the data member from the VARIANT and dereference it to make an indirect call, since the process of deleting an IDispatch object involves calling the IDispatch::Release() function. Omission of the VariantInit() function creates a condition not unlike the memory management analog of freeing a block of memory without first allocating it, with two key differences:

1. Double VariantClear() is not the same as double free() – since VariantClear() sets the VARIANT type to VT_EMPTY, any subsequent calls to VariantClear() for the same VARIANT will have no effect, and
2. Omission of VariantInit() is more likely than free() without malloc(), because the code will still seemingly work correctly most of the time, even if the vulnerable code is exercised.

This class of mistakes is really an uninitialized variable problem, but is included in this section because it results in a form of type confusion, with the additional caveat that the attacker needs to prime the appropriate memory area with useful data rather than specifying it directly. That is, the exploitability of these problems is very dependent on the residual data contained in the memory where the VARIANT was allocated. Under some conditions, this data is under the control of the attacker, while in other cases, the attacker simply needs to get lucky.

An example VariantInit() omission vulnerability is shown below.

```
HRESULT MyFunc(IStream* pStream)
{
    VARIANT var;
```

```
    IDispatch* pDisp;
    HRESULT hr;

    var.vt = VT_DISPATCH;

    hr = pStream->Read(pDisp, sizeof(IDispatch *), NULL);

    if(FAILED(hr)) {
      VariantClear(&var);
      return hr;
    }

    . . .

    return hr;
}
```

As can be seen, a VARIANT located on the stack is manually initialized with the type VT_DISPATCH, and is presumably filled out with a pointer to an IDispatch interface after data has been successfully read from the source stream. However, if the IStream::Read() operation fails, the VARIANT is cleared, resulting in manipulating uninitialized stack data as if it pointed to an IDispatch interface.

Although this seems like a relatively unlikely mistake to make, there are sometimes variations of the vulnerable code path that are slightly more subtle. One such example occurs when copying data between VARIANTs using the VariantCopy() function. The VariantCopy() function clears the destination VARIANT parameter before copying anything to it. Therefore, the destination parameter passed to VariantCopy() must be cleared first as well. The code below demonstrates a vulnerable condition with the same exploitability constraints as the previous example.

```
HRESULT MyFunc(IStream* pStream)
{
    VARIANT srcVar;
    VARIANT dstVar;
    IDispatch* pDisp;
    HRESULT hr;

    srcVar.vt = VT_DISPATCH;
    dstVar.vt = VT_DISPATCH;

    hr = pStream->Read(pDisp, sizeof(IDispatch *), NULL);

    if(FAILED(hr)) {
      //VariantClear(&var);
      return hr;
    }
    else {
        srcVar.pdispVal = pDisp;
        hr = VariantCopy(&dstVar, &srcVar);
```

```
    }

    return hr;
}
```

Similar problems also exist in other VARIANT API functions, most notably the VariantChangeType()/VariantChangeTypeEx() functions. These functions will use VariantClear() in some but not all conversion cases. The rules for when VariantClear() is called on the destination value are for the most part intuitive; they occur when:

- An invalid conversion attempt is not encountered (ie. not converting between two incompatible types), and
- The VariantClear() won't cause problems when the source and destination VARIANT are the same, such as converting from a VT_UNKNOWN -> VT_DISPATCH.

In terms of auditing for vulnerabilities, any conversion where the destination parameter is uninitialized should be viewed critically. For example, consider the following code.

```
BSTR *ExtractStringFromVariant(VARIANT *var)
{
   VARIANT dstVar;
   HRESULT hr;
   BSTR *res;

   if(var->vt == VT_BSTR)
      return SysAllocString(var->bstrVal);

   else {
     hr = VariantChangeType(&dstVar, var, 0, VT_BSTR);
    if(FAILED(hr))
       return NULL;
   }

   res = SysAllocString(dstVar.bstrVal);
   VariantClear(&dstVar);

   return res;
}
```

Here we see a similar construct to the previous examples, except this time using VariantChangeType(). The following requirements for exploitation exist:

1. The destination VARiANT is uninitiliazed, and
2. A conversion from a regular type to VT_BSTR will result in VariantClear() on the destination VARIANT (such as VT_I4 -> VT_BSTR)

As mentioned previously, successful exploitation of the above vulnerability would require the attacker to be able to influence the stack so that the uninitiliazed destination VARIANT had useful data in it, such as having a type of VT_DISPATCH and some sort of valid pointer as the value.

Mozilla Type Confusion Vulnerabilities: NPAPI

Most non-IE browsers implement the NPAPI for plugin interaction, which in turn utilizes the NPRuntime to expose scriptable objects to scripting languages. The API for passing variables to and from plugins is much more simple than those utilized by COM and IE, resulting in a reduced attack surface. However, the NPRuntime still presents interesting opportunities to attackers, as it lends itself to misuse that can result in type confusion vulnerabilities similar to those we have seen with VARIANTs. This section explores how type confusion vulnerabilities can occur in the context of NPRuntime scriptable objects. This discussion is applicable to all browsers that implement the NPAPI and expose NPRuntime functionality to web content.

NPAPI Type Confusion Attacks I: Type Validation

One of the key differences between the NPRuntime and the COM VARIANT passing we have already looked at is that NPRuntime does not perform any type coercion or validation on NPVariants received from scripting hosts. Recall from our previous discussion of NPRuntime how a plugin accesses an NPVariant; by using one of the NPVARIANT_TO_XXX() macros. These macros do nothing other than access a member of the union data structure contained within an NPVariant – the onus is on the plugin developer to ensure that the variant is of the correct type by using the corresponding NPVARIANT_IS_XXX() macro. A plugin that correctly handles NPVariant arguments might look like this:

```
bool SetProperty(NPObject *obj, NPIdentifier name, const NPVariant *variant)
{
      if(name == kTestIdent)
      {
            if(!NPVARIANT_IS_INT32(*variant))
                  return false;

            gTest = NPVARIANT_TO_INT32(*variant);
            return true;
      }
      return false;
}
```

This example is the expected algorithm to manipulate NPVariants – a check for the correct type followed by the data access. Each time a function receives an NPVariant, this type check must be performed before processing its data. The absence of the initial check renders the code vulnerable to type confusion problems. To illustrate problems with regard to failing to perform the initial check, consider the following code taken from Google's "Native Client" plugin:

```
bool Plugin::SetProperty(NPObject* obj,
                         NPIdentifier name,
                         const NPVariant* variant) {
  Plugin* plugin = reinterpret_cast<Plugin*>(obj);

  if (kHeightIdent == name) {
    plugin->height_ = NPVARIANT_TO_INT32(*variant);
    return true;
```

This function sets the "height" property of the object in question, however fails to ensure that the NPVariant being manipulated is an integer. It would be possible for an attacker to pass a string or object in as the height parameter instead of an integer, resulting in a pointer being confused as an integer. This code would most likely result in an information leak vulnerability that discloses a pointer when the attacker reads back the height property at a later time. Obviously, the opposite situation is potentially more dangerous – one in which an attacker can supply an integer in place of a pointer. Depending on how the pointer is manipulated, this situation can either lead to a more expansive information leak, or a memory corruption vulnerability.

One slightly more subtle variation on this attack is one in which an NPVariant is validated to be an NPObject, and the plugin attempts to cast the generic NPObject to a specific type of object. The NPAPI runtime lacks API functions that allow you to determine if it's safe to perform this object conversion, so this construct is nearly always going to lend itself to exploitation. Returning to Native Client, consider the following code.

```
static bool GetHandle(struct NaClDesc** v, NPVariant var) {
  if (NPVARIANT_IS_OBJECT(var)) {
    NPObject* obj = NPVARIANT_TO_OBJECT(var);
    UnknownHandle* handle = reinterpret_cast<UnknownHandle*>(obj);

    *v = handle->desc();
    return true;
  } else {
    return false;
  }
}
```

This code is responsible for receiving a "handle" object from JavaScript. Handle objects are a specific specialization of scriptable objects that is implemented by Native Client for communication with their back end. The code correctly validates that the received NPVariant is indeed a JavaScript object using the NPVARIANT_IS_OBJECT() macro. However, they subsequently cast the received NPObject pointer to an UnknownHandle pointer. Since an attacker may supply an arbitrary JavaScript object here, it is possible to perform a type confusion attack where any random NPObject is confused with the UnknownHandle object. The most likely outcome of type confusion vulnerabilities of this nature is arbitrary code execution.

One thing worth mentioning here is that the inputs to NPObject functions are not necessarily the only ways to supply potentially mis-typed objects. As mentioned in the technology overview, the NPN_GetProperty() function is used to retrieve objects from the DOM hierarchy. Since these objects are subject to scripting control, manipulating objects visible in the DOM can be an entry point for performing similar attacks to those described here.

NPAPI Type Confusion Attacks II: Parameter Count Validation

The Invoke() and InvokeDefault() methods that are exposed by an NPObject are required to verify the number and type of parameters that are passed to them, against the correct number and type of arguments for the method identified by the NPIdentifier parameter. For both functions, verification of the number of arguments is achieved by simply ensuring the argc parameter contains the correct value. Although it is a less common mistake to make, the plugin developer is required to verify the argc

parameter for every callable function within Invoke() and InvokeDefault() – it is not automatically verified. Failure to verify it can lead to situations where invalid array indexes are used to retrieve argument parameters. Some vulnerable sample code is shown below:

```
bool Invoke(NPObject *obj, NPIdentifier name, const NPVariant *args, uint32_t
argCount, NPVariant *result)
{
      if(name == kTestFuncName)
      {
            if(argCount != 2 &&
            (!NPVARIANT_IS_INT32(args[0]) || !NPVARIANT_IS_STRING(args[1])))
                  return false;

            unsigned int length = NPVARIANT_TO_INT32(args[0]);
            char *buffer = ExtractString(args[1]);

            ... more code ...
      }
}
```

The code above is well-intentioned – it attempts to check both the parameter count as well as the types of each parameter. However, there is a problem in the check: the logical and (&&) operator is used where the logical OR (||) should have been. As such, it is possible to pass the verification and have the processing code executed with a number of parameters that is different than the number it expects. If only a single parameter is passed, out-of-bounds memory will be accessed for any manipulation of the second element of the args array.

The previous code construct results in an uninitialized variable being used and it could be argued that it is more properly categorized as an uninitialized variable problem; however, this erroneous behavior stems from the relative ambiguity of NPVariant parameters compared to native variable types. Therefore, it is included in this section because it stems from type ambiguity and because the semantics of this problem are similar to those described in the previous section.

Interoperability Attacks III: Trust in Executable Modules

Interoperability imposes unique requirements on the execution environment. First, an application needs to ensure that the components it instantiates adhere to the security requirements of the application. Ensuring this fact is difficult because components written for interoperability don't require a specific environment; therefore, they will largely be ignorant of any environment-specific security standards that may be required. Indeed, when reviewing Microsoft's security surrounding COM it is easy to postulate that the fractured security architecture is a consequence of this complexity.

Further compounding this problem is the fact that an interoperability component may require the use of one or more sub-components. Assuming that an application has a method to completely ensure that an interoperability component is safe to run in the application's context, the application may still be completely unaware of which sub-components the vetted component brings into the execution environment. Either the application environment or the super-component must be responsible for ensuring the sub-components are trusted for the execution environment.

Transitive trust is a term we use to denote the condition where a component has the ability to extend the trust it has been granted by a host application, to objects the component may rely on, at the component's discretion. In the context of web browsers, the authorization models employed in practice are flat; that is, only a super-component undergoes explicit authorization checks by the host application. Figure 23 depicts an example chain of trust.

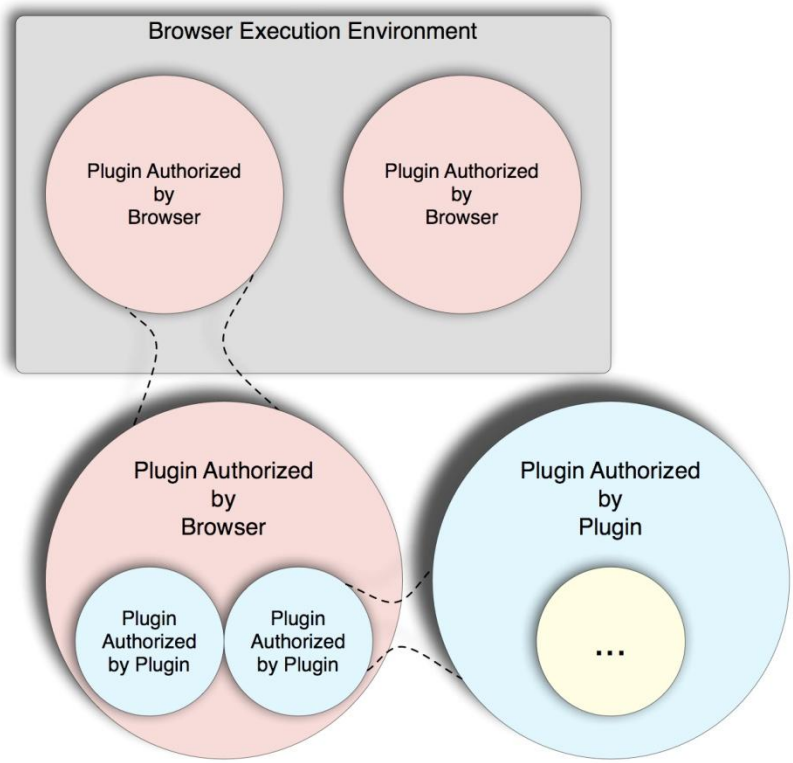


Figure 23: Image depicting a chain of trust for browsers and plugins.

As the picture indicates, the impetus of enforcing the security model is placed squarely upon the super-component. This model where super-objects may rely on sub-objects creates a chain of trust where each link in the chain is verified by objects from different code-bases, using potentially disparate strategies, possibly possessing a limited idea of the trust models they are inheriting; hence, they may not enforce the model with complete fidelity. Furthermore, retro-fitted security features added to the host application over time can often be undermined by plugins or components that were not originally designed to be compliant with the new restrictions. As such, new security features can often be conveniently bypassed by attackers utilizing plugin features creatively. This section will present several attacks that the authors uncovered that fall into the category of transitive trust – the utilization of plugin or component features to undermine security features built in to the web browser.

Transitive Trust Vulnerabilities I - Persistent Objects

This paper has already discussed the implementation and use of persistent COM objects at length, and some of the challenges that they present in terms of security. In addition to the issues already discussed, persistent objects provide the attacker with the ability to cause objects to load property values, sometimes even values of arbitrary types. The following sections will explore the implication this ability has with respect to transitive trust vulnerabilities, ultimately outlining methods of bypassing the security features Internet Explorer relies upon to safely deliver web content.

Transitive Trust Vulnerabilities - Bypassing Control Authorizations (The Highlander Bit)

As discussed in section two of this paper, IE implements various controls to restrict which ActiveX objects may be instantiated in the context of the browser, and the types of warnings the user is presented with before a control is authorized for the browser context. As we previously described, for an object to be deemed safe to load in Internet Explorer's execution environment, it needs to be marked as safe for scripting and/or safe for initialization, the killbit for the object in question must not be set, and finally, the control must be approved to run in the domain.

The preapproved list is populated when Internet Explorer is first installed, and any other changes to this list will result from user customization. This list limits the number of controls an attacker can leverage without Internet Explorer notifying the target that the content might undermine the browser's security. Additionally, Microsoft has been distributing cumulative killbit settings over time as part of their monthly security bundles to ensure that a large number of exploitable controls aren't loadable in the context of IE. In fact, in several instances, Microsoft opted to disable controls through adding killbits rather than attempting to fix the underlying vulnerabilities as they were discovered. It is easy to see that bypassing these authorizations is quite desirable from an attacker's viewpoint, and Object persistence lends itself to achieving this goal.

There are several controls available on a typical Windows machine that can be initialized without prompting the user. Since most Automation controls use the default ATL implementation of IPersistStream to resurrect objects into memory, we will consider the Load() method from this

implementation. Most of the work for restoring an object is actually performed by CComVariant::ReadFromStream(), whose implementation is partially shown:

```
HRESULT VariantCopy(VARIANTARG *pvargDest, VARIANTARG *pvargSrc);
HRESULT VariantCopyInd(VARIANTARG *pvargDest, VARIANTARG *pvargSrc);

inline HRESULT CComVariant::ReadFromStream(IStream* pStream)
{
      ATLASSERT(pStream != NULL);
      if(pStream == NULL)
            return E_INVALIDARG;

      HRESULT hr;
      hr = VariantClear(this);
      if (FAILED(hr))
            return hr;
      VARTYPE vtRead = VT_EMPTY;
      ULONG cbRead = 0;
      hr = pStream->Read(&vtRead, sizeof(VARTYPE), &cbRead);
      if (hr == S_FALSE || (cbRead != sizeof(VARTYPE) && hr == S_OK))
            hr = E_FAIL;
      if (FAILED(hr))
            return hr;

      vt = vtRead;
      cbRead = 0;
      switch (vtRead)
      {
      case VT_UNKNOWN:
      case VT_DISPATCH:
            {
                  punkVal = NULL;
                  hr = OleLoadFromStream(pStream,
                        (vtRead == VT_UNKNOWN) ?
                        __uuidof(IUnknown) : __uuidof(IDispatch),
                        (void**)&punkVal);
                  // If IPictureDisp or IFontDisp property is not set,
                  // OleLoadFromStream() will
                  // return REGDB_E_CLASSNOTREG.

                  if (hr == REGDB_E_CLASSNOTREG)
                        hr = S_OK;
                  return hr;
            }
      case VT_UI1:
      case VT_I1:
            cbRead = sizeof(BYTE);
            break;

      ... more object types ...
```

```
      default:
            break;
      }

      ... more code ...
}
```

As can be seen by reviewing the above code, when a VT_DISPATCH or VT_UNKNOWN object is read, the IStream is passed to OleLoadFromStream() to read the subordinate object into memory. Pseudocode for OleLoadFromStream(), exported from ole32.dll is shown below:

```
HRESULT __stdcall OleLoadFromStream(LPSTREAM pStm, const IID *const
iidInterface, LPVOID *ppvObj)
{
      IPersistStream *pIPersistStream;
      IUnknown *pIUnknown;
      CLSID clsidControl;
      HRESULT hrValue;

            *ppbObj = NULL;
            hrValue = ReadClassStm(pStm, &clsidControl);
            if(hrValue != ERROR_SUCCESS)
                  return(hrValue);
            hrValue = CoCreateInstance(&clsidControl, NULL, \
                        CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER | \
                        CLSCTX_REMOTE_SERVER | CLSCTX_NO_CODE_DOWNLOAD, \
                        iidInterface, &pIUnknown);
            if(hrValue != ERROR_SUCCESS)
                  return(hrValue);
            hrValue = pIUnknown->QueryInterface(CLSID_IPersistStream, \
                        &pIPersistStream);
            if(hrValue != ERROR_SUCCESS)
                  goto CleanupIUnknown;
            hrValue = pIPersistStream->Load(pStm);
            pIPersistStream->Release();
            if(hrValue != ERROR_SUCCESS)
                  goto CleanupIUnknown;
            hrValue = pIUnknown->QueryInterface(iidInterface, ppvObj);

CleanupIUnknown:
            pIUnknown->Release();
            return(hrValue);
}
```

As can be seen above, the OleLoadFromStream() function will call CoCreateInstance() using a CLSID that is provided in the IStream, and subsequently initialize the control with persistence data. If attackers are

able to supply this persistence data, then they can use this code to load any arbitrary COM object and supply the object with persistence data. Most importantly, there is no functionality to determine if the subordinate control meets the security requirements of the host application – including the killbit status of the control, and any logic that might request approval from the user. It should be noted that at the time of this writing, using this method appears to only provides access to the Load() method in the IPersistStream interface of the control loaded from persistence data. However, this capability is perfectly sufficient to provide a vector, unhampered by security restrictions, that allows access to vulnerabilities present in persistence routines, and numerous previously disclosed vulnerabilities. Table 24 lists a small sampling of controls that can be reached and have been reported to trigger a vulnerability merely on object instantiation, or by processing persistence data.

GUID	File
0955AC62-BF2E-4CBA-A2B9-A63F772D46CF	Msvidctl.dll
47C6C527-6204-4F91-849D-66E234DEE015	Srchui.dll
35CEC8A3-2BE6-11D2-8773-92E220524153	Stobject.dll
730F6CDC-2C86-11D2-8773-92E220524153	Stobject.dll
2C10A98F-D64F-43B4-BED6-DD0E1BF2074C	Vdt70.dll
6F9F3481-84DD-4B14-B09C-6B4288ECCDE8	Vdt70.dll
8E26BFC1-AFD6-11CF-BFFC-00AA003CFDFC	Vmhelper.dll
F0975AFE-5C7F-11D2-8B74-00104B2AFB41	Wbemads.dll

Table 24: List of vulnerable controls that can be reached through the Highlander bit

Starting with ATL version 2, which was distributed with Visual Studio 97, up to and including ATL version 8.0, distributed with Visual Studio 2005, there were no mechanisms for granular control of the property type that was to be read from the stream for any macro other than PROP_DATA_ENTRY; therefore, most properties the control read from the stream could be read as a VT_DISPATCH or VT_UNKNOWN variant. In ATL version 9.0, distributed with Visual Studio 2008, property entry macros that did not specify a type were declared deprecated and CComVariant::ReadFromStream() requires that the type read from the stream is equivalent to the type specified in the macro unless the type specified is equal to VT_EMPTY. However, several third party controls (most notably Macromedia's Flash control) have property entries that specify a VT_DISPATCH type, and will still allow this vector. Furthermore, several Microsoft controls that implement custom Load() methods provide the ability for attackers to load arbitrary objects as well. The following example code is part of the IPersistStream implementation for Microsoft's ComponentTypes control.

```
HRESULT __stdcall CComponentTypes::Load(struct IStream *pStm)
{
	HRESULT hrVal;
	ULONG ulRead;
	long lCntComponents;
	long lIndexComponent;

	hrVal = pStm->Read(&lCntComponents, sizeof(lCntComponents), &ulRead);
	if(hrVal < ERROR_SUCCESS)
		return(hrVal);
```

```
      if(ulRead != sizeof(lCntComponents))
            return(E_UNEXPECTED);
      for(  lIndexComponent = 0; \
            lIndexComponent < lCntComponents;  \
            lIndexComponent++)
      {
            GUID2 ReadGuid;
            hrVal = pStm->Read(&ReadGuid, sizeof(ReadGuid), &ulRead);
            if(hrVal < ERROR_SUCCESS)
                  return(hrVal);
            CComQIPtr<IPropertyBag,
                 &__s_GUID const_GUID_55272a00_42cb_11ce_8135_00aa004bb851> \
                 myControl;

            hrVal = CoCreateInstance(&ReadGuid, NULL, CLSCTX_INPROC_SERVER| \
                        CLSCTX_INPROC_HANDLER, IID_IPersistStreamInit, \
                        &myControl);
            if(hrVal < ERROR_SUCCESS)
                  return(hrVal);
            hrVal = myControl.Load(pStm);
            if(hrVal < ERROR_SUCCESS)
                  return(hrVal);
            ...
```

The code reads an integer specifying the number of controls in the stream. Next, it will read in a class ID and attempt to load the control from the persistent stream. It will repeat the last step until it encounters an error, or it has read a number of controls equivalent to the first integer value in the stream. Again, attackers can specify the stream this control reads from and the control doesn't perform any authorization checks on this control before loading it with attacker-supplied persistence data.

Section IV: Conclusion

Interoperability provides applications with the benefit of offering increased flexibility by utilizing pluggable components. However, the cost of this flexibility from a security standpoint is one that is often overlooked to a large extent. We have presented attacks that target the interoperability functionality itself – from marshalling and management of data objects across module boundaries to exploiting the extension of trust given to plugins or core components. Additionally, we have shown that these areas are more susceptible to unique bug classes that have been paid modest or no real attention in the past. Attackers that wish to target applications that employ interoperability to communicate between unrelated components could use such techniques to uncover subtle flaws in data manipulation code or defeat counter-measures designed to thwart security breaches through the exploitation of relaxed trust boundaries. Further research in interoperability will likely yield further unique exploitation scenarios, especially in the area of transitive trust. This is due to security barriers as well as new components being constantly added to rich applications such as web browsers.