Attacking Interoperability
Hack In the Box (HITB) Malaysia, October 2009

Mark Dowd (mark.dowd@gmail.com)
Ryan Smith (ryan@hustlelabs.com)
David Dewey (dewey@us.ibm.com)
INTRODUCTION
What is Interoperability?
- The glue that allows different software to talk to each other
  - Like a translator – eg. Russian <-> Arabic
- Required to allow any two disparate components to communicate directly
  - Javascript <-> DOM, Java <-> Javascript, .NET <-> Javascript…
- **Thesis**
  - The added complexity of expanded interoperability creates subtle, new, and interesting opportunities for attackers
    - Large, nuanced attack surface
    - Speech targets browser components, but ideas applicable elsewhere

- **Overview**
  - Part I – Technology Overview
  - Part II – Attack Surface
  - Part III – Bug Classes (and practical attacks)

- **We will discuss subtle vulnerabilities in Microsoft’s premier browser**
TECHNOLOGY OVERVIEW
INTERNET EXPLORER 8
Internet Explorer 8 provides additional security on Vista/Win7
- “Loosely Coupled”
- Architecture restructuring from previous versions
- Provides a sandbox (“Protected Mode”) for browsing by utilizing Vista/Win7 “Integrity Levels”

Runs at 3 different Integrity levels
- Low Integrity (default for “Internet Zone” browsing)
- Medium Integrity (default for “Intranet Zone” browsing” and user broker)
- High Integrity (Installation broker)

Protects the OS from the browser
What are “Integrity Levels”, anyway?
- Mandatory Integrity Control (MIC) ensures securable objects not writeable by lower integrity processes
- Integrity SIDs attached to the user’s access token (default: medium)
- Securable objects have Integrity SID Access Control Entry (ACE) attached to the SACL
  - If no SACL is present, it is assumed to be “medium”
- SIDs of calling process and requested object compared before DACLs

Programs executed with the same integrity as the caller
- Exception: If the executable’s SID is “low”, it is executed as such

Also prevents “shatter attacks” by filtering Window messages (UIPI)
- Can only write to low integrity locations
- Can only write to low integrity registry locations (HKEY_CURRENT_USER\Software\LowRegistry)

- Programs can be executed at elevated privileges
  - Registry key exists with list of allowed programs (HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\Low Rights\ElevationPolicy)
Application launch is dictated by ‘Policy’ value

<table>
<thead>
<tr>
<th>Policy Value</th>
<th>Meaning</th>
<th>Integrity Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Do not launch application</td>
<td>N/A</td>
</tr>
<tr>
<td>1</td>
<td>Silently launch application</td>
<td>Low</td>
</tr>
<tr>
<td>2</td>
<td>Prompt before launching</td>
<td>Medium</td>
</tr>
<tr>
<td>3</td>
<td>Silently launch application</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Similar policies exist for other low integrity requirements
- Allowing Drag & Drop (HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\Low Rights\DragDrop)
- Allowing DLLs to be launched via rundll32.exe (HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\Low Rights\RunDII32Policy)
  - Only allows low integrity launching
ACTIVEX
Everyone knows what ActiveX controls are
- Registered in HKCR\CLSID\{<CLSID>\}

Functional restrictions
- SFI: Instantiation from persistent COM stream
- SFS: Control can be scripted

Existential Restrictions
- SiteLock
- System-wide killbits
- IE8: Per-user killbits

Restrictions requiring extra operations
- IE7: System-wide white list of controls
- IE8: Per-user and per-domain white list of controls
ActiveX controls and scriptable objects are IDispatch / IDispatchEx COM objects

- Self-publishing
- Methods / properties called via Invoke() function using DispID
- Parameters passed in DISPPARAMS structure
- Essentially array of VARIANTARGS

```c
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;
```
VARIANT data structure used to represent data types
- Data structure with type variable (vt) and value variable (union)
- Types consist of basic type (0 -> 0xFFF) + possible modifiers (0x1000+)
- VT_BYREF modifier not mutually exclusive with other modifiers

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Value</th>
<th>Union Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_EMPTY</td>
<td>0x0000</td>
<td>Undefined</td>
</tr>
<tr>
<td>VT_NULL</td>
<td>0x0001</td>
<td>NULL value</td>
</tr>
<tr>
<td>VT_I4</td>
<td>0x0003</td>
<td>Signed (4-byte) integer</td>
</tr>
<tr>
<td>VT_BSTR</td>
<td>0x0008</td>
<td>String; Pointer to a BSTR</td>
</tr>
<tr>
<td>VT_DISPATCH</td>
<td>0x0009</td>
<td>Pointer to an IDispatch interface (automation object)</td>
</tr>
<tr>
<td>VT_BOOL</td>
<td>0x000B</td>
<td>Boolean (2-byte short)</td>
</tr>
<tr>
<td>VT_VARIANT</td>
<td>0x000C</td>
<td>Pointer to another VARIANT</td>
</tr>
<tr>
<td>VT_UNKNOWN</td>
<td>0x000D</td>
<td>Pointer to an IUnknown interface (any COM object)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modifier Name</th>
<th>Modifier Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_VECTOR</td>
<td>0x1000</td>
<td>Value points to a simple counted array (Rarely used)</td>
</tr>
<tr>
<td>VT_ARRAY</td>
<td>0x2000</td>
<td>Value points to a SAFEARRAY structure</td>
</tr>
<tr>
<td>VT_BYREF</td>
<td>0x4000</td>
<td>Value points to base type, instead of containing a literal of the base type</td>
</tr>
</tbody>
</table>
COM serialization

- Accessed via COM interfaces
  - IStream and IStorage
- Represents a file / memory stream / etc
- Support persistence by implementing one of the IPersist interfaces
### COM persist streams

- Binary data representing object properties
- Interpretation depends on the IPersist*::Load() method
- Most use ATL IPersistStream::Load()
- Programmer defines a property map
  - BEGIN_PROPERTY_MAP(), BEGIN_PROP_MAP(), PROP_ENTRY(), PROP_ENTRY_EX(), PROP_DATA_ENTRY(), etc

### COM persistent streams embeddable in IE

- Property Bags: <PARAM> tags
- Binary files retrieved from “data” parameter of <OBJECT> tag
  - .ICA, .STM, .ODS extensions -> IPersistStream
  - Otherwise, query for IPersist*
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()
};

<table>
<thead>
<tr>
<th>Offset</th>
<th>Hexadecimal representation of bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>00 09 00 00</td>
<td>Version nine of the ATL</td>
</tr>
<tr>
<td>0x04</td>
<td>00 01 00 00</td>
<td>The _cx value is 256</td>
</tr>
<tr>
<td>0x08</td>
<td>00 01 00 00</td>
<td>The _cy value is 256</td>
</tr>
<tr>
<td>0x0C</td>
<td>08 00</td>
<td>NameFirst is stored as a VT_BSTR</td>
</tr>
<tr>
<td>0x0E</td>
<td>0C 00 00 00</td>
<td>NameFirst is 12 characters long</td>
</tr>
<tr>
<td>0x12</td>
<td>46 00 69 00 72 00 73 00 74 00 00</td>
<td>NameFirst is equivalent to &quot;First&quot;</td>
</tr>
<tr>
<td>0x1E</td>
<td>0A 00 00 00</td>
<td>NameLast is 10 bytes long</td>
</tr>
<tr>
<td>0x22</td>
<td>4C 00 61 00 73 00 74 00 00 00</td>
<td>NameLast is equivalent to &quot;Last&quot;</td>
</tr>
</tbody>
</table>
DOM Implementation
- Code that implements DOM functionality
- Code that marshals data between DOM <-> other languages

ActiveX controls
- Safe for Scripting (SFS) persistence functionality (IPersist*)
- Exposed functions that perform VARIANT or object manipulation

Language Runtimes
- Native functionality
- Marshalling

Trust boundaries
- Security mechanism implementations
- Components that (potentially) operate outside security mechanism controls
DOM IMPLEMENTATION
- **DOM is a large native code base**
  - Functionality exposed by the DOM implemented natively
  - Marshalling routines for input / output objects
  - All implemented within MSHTML.DLL

- **Extensive amount of functionality reachable via scripting**
  - Base HTML element methods
  - Context-specific methods
  - Gettable/Settable properties

- **Objects utilize COM extensively**
  - Methods, properties, and events exposed through COM interfaces
  - Objects from the DOM expose IDispatchEx for manipulation
  - Not unlike ActiveX objects
Properties of an object published via Class Descriptor
- All extend from the C++ CBase class (not externally COM-visible)
- Each class has a descriptor available via CBase::GetClassDesc()
- Contains information about how the object may be manipulated

Class descriptors contain property / method info
- Descriptors indirectly accessible via an objects class descriptor
- Property descriptors include marshaling and implementation information
- Versioning, name hashes, DispIDs, enum info etc also present
IE DOM Class Descriptors

Class Descriptor

HDL Descriptor

String Aggregate Table

AssocVTTable (HashTable for VTable Descriptors)

String Table

VTTable Descriptor

Property Descriptor

VTTable Descriptor

Property Descriptor

VTTable Descriptor

Property Descriptor

VTTable Descriptor

Property Descriptor

VTTable Descriptor

Property Descriptor

AssocVTTable (HashTable for VTable Descriptors)
- Flag field in the descriptor provides relevant information
  - Offset into a global IID array for the IID to query the object for
  - Offset into the returned VTable where native implementation resides

- Special case: if iid_offset is 0, then a default IID is used from the class descriptor

```c
typedef struct _PROPERTYDESC
{
    struct _PROPERTYDESC *pNext;   // offset 0x00
    WCHAR *pCustomName;            // offset 0x04
    WCHAR *pName;                  // offset 0x08
    DWORD defaultValue;            // offset 0x0C
    DWORD hash;                    // offset 0x10
    DWORD flags;                   // offset 0x14 "300"
    DWORD dispId;                  // offset 0x18 "44E"
    DWORD off1C;                   // offset 0x1C "0"
    USHORT marshalerIdx;           // offset 0x20
    DWORD off24;                   // offset 0x24
    USHORT off28;                  // offset 0x28
    USHORT off2A;                  // offset 0x2A
    LPVOID pEnum;                  // offset 0x2C - struct ENUMDESC
} PROPERTYDESC, *LPPROPERTYDESC;
```
• Marshalling for the DOM is similar to ActiveX
  - Parameters passed in DISPPARAMs structure
  - Layer exists for type checking / conversion of parameters for called function

• Implemented as a series of marshalling routines
  - Method naming convention: Method_rettype_param1type_param2type()
  - Get/Set naming convention: G(S)_type()
  - Stored in a large array
- **Methods wrap native function**
  - Invoked from CBase::ContextInvokeEx()
  - Perform type checking (and conversion if necessary)
  - Release created objects if an error occurs
  - Calls the native method
  - Stores return value and cleans up

- **How is the appropriate marshalling method selected?**
  - Indicated by “marshalerldx” member in property descriptor
  - Used as an index in to marshalling array

- **Result: We can map exposed functionality to native code automatically**
  - Find marshaling routines using marshalerldx value
  - Find native function using previously mentioned techniques
  - Marshaling method used infers data types passed to and from native method
• Demo
ACTIVEX CONTROLS
- ActiveX scripting interface
  - Methods exposed via IDispatch
  - Described by type library
  - Well-known entrypoints (targeted often)

- Safe for Initialization surface also interesting
  - IPersist* interface
    - Load() methods parse untrusted serialized COM data
  - Often less explored than scripting interfaces
- **ATL standard implementations often used**
  - Exposes large parts of ATL core as attack surface
  - Extent of attack surface varies according to property map

- **Property maps extend the attack surface**
  - Definition of properties creates potential for bugs that otherwise couldn’t be triggered
    - Loose Typing
    - Bypassing IDispatch
    - Embedded objects
  - We will show the impact of all of these shortly..
LANGUAGE RUNTIMES
Language Runtimes contain lots of native functionality
- Major scripting engines: VBScript and JavaScript
- Native implementations of the core functionality exposed
- Processing untrusted inputs

Marshaling is required for inputs
- Much like the DOM
- Marshaling seems to be dispersed throughout the code
- More chances of mistakes this way

Enumerating the attack surface – locating native functionality + interoperability
- Bonus: learning this stuff is probably useful for exploitation…. 
Both languages internally manage data with similar structures
- NameTbl class for objects
- SYM for representing symbols
- VAR class for wrapping VARIANT values
- VVAL class wraps named variables (VAR + DispID etc)

Enumeration of both binaries can be done similarly by manipulating these structures
- Key difference: Javascript’s functionality is tied up in built-in objects, VBScript’s isn’t
- Native methods are registered within relevant object (NameTbl class)
  - Global methods registered within GlobalBinder::EnsureBuiltin()
  - Object methods registered within <ObjectClass>::EnsureBuiltin()

- Registration method varies
  - NameTbl::AddNativeMethod() used for most situations
  - Direct registration for functions that calculate properties (using type <ClassObject>::CreateVval())
  - Second type common for registering property functions that need to be calculated at runtime

- Standard interfaces for functions
  - Member functions (including global objects): CSession *pSession, VAR *pThis, VAR *pRet, int argCount, VAR *pArgvArray
  - Property functions: VAR *pThis, VAR *pRet, int cmd
    - Cmd = 0 (get property), cmd = 1 (set property)
VBScript Native functionality is clearly exported
- All native methods stored in “StaticEntryPoint” data structures
- Available contiguously in the .data section of vbscript.dll
- Structure has Vtable followed by method
- Methods all registered to the runtime in GlobalBinder::EnsureBuiltin()

All entrypoints adhere to the same interface
- Three inputs: return type, parameter count, parameter array
- Inputs / outputs stored in VAR classes (a VARIANT wrapper)

Each entry point does it’s own ad-hoc marshaling
- Generally uses functions like PvarConvert(), which throws exceptions on type conversion error
- Parameter count must also be manually verified
Demo
- Interoperability layers affected by standard bug classes
  - Buffer overflows (Boring)
  - Memory Corruption (Boring)

- Additional complexities
  - Language agnostic variable representation
  - Lifespan of data
  - Security models

- Unique challenges result in unique vulnerabilities
  - Type confusion vulnerabilities
  - Transitive trust vulnerabilities
TYPE CONFUSION
- Interoperability requires language agnostic data representation
  - Contrived types
  - COM: VARIANT/VARIANTARG

- Contrived types require careful programming
  - Big opportunity!

- Vulnerabilities occur when one data type is mistaken for another.
  - Unions
Unions

- Same memory space – different types
- Useful for conserving memory and abstracting data
- No errors or warnings at compile time when accessing incorrect type

Programmer must keep track of the appropriate member
- High-level identifiers keep track
- Easy to get wrong (APIs somewhat unintuitive at times)
- **VARIANT Type Confusion I - Permissive property maps**
  - Property maps tell the ATL what types to serialize / resurrect
  - During resurrection, the array is traversed

- **Some property map entry macros require a specific variant type**
  - PROP_ENTRY_TYPE()
  - PROP_ENTRY_TYPE_EX()
  - PROP_DATA_ENTRY()

- **Others are more permissive**
  - PROP_ENTRY()
  - PROP_ENTRY_EX()
  - Any of the less permissive ones if passed VT_EMPTY as the type
- **VARIANT Type Confusion II - Initialization Errors**
  - Operating on VARIANTs that are partially or totally uninitialized
  - VARIANT API lends itself to these types of problems

- **VariantClear() on uninitialized VARIANTs considered dangerous!**
  - Need to call VariantInit() first (or otherwise set vt to VT_EMPTY)
  - Easy to forget to initialize them!

- **VARIANT API functions often VariantClear() their dst parameters**
  - VariantCopy()
  - VariantCopyInd()
  - VariantChangeType()
  - VariantChangeTypeEx()
Example II – Uninitialized VARIANTs

- var never initialized with VariantInit()
- If read fails, VariantClear() called

```c
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);
        VariantClear(&var);
    }
    
    return hr;
}
```
Example III – VariantCopy Example

- VariantCopy() calls VariantClear() on dstVar
- dstVar is uninitialized

```c
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;
}
```
VARIANT Type Confusion III – Misinterpreting types
- vt contains basic type + modifiers

Type interpretation is susceptible to subtle errors
- Masking off all the modifiers
  - Just operating on the basic type
- Action based on a specific modifier
  - Eg. VT_ARRAY is set, doesn’t mean the type is a SAFEARRAY!
- Masking off specific modifiers
  - Losing information is bad
Example IV – IEs core DOM marshalling
- VT_ARRAY modifier is masked off! (VT_TYPEMASK = 0xFFF)

```c
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)
        {
            ... Try type conversion, die on failure ...
        }
    }
    switch(vt)
    {
        case VT_DISPATCH:
            *(PDISPATCH)outVar = pSrcVar->pdispVal;
            break;
        ... Other types dealt with here ...
    }
}```
- **VARIANT Type Confusion III – Direct type manipulation**
  - Setting the vt directly
  - Calling an API function, failure to check if it succeeds
  - Mainly a result of VariantChangeType() / VariantChangeTypeEx() failing

- **Setting the type manually can have significant consequences**
  - Type confusion if error isn’t detected
  - Possible type confusion even if error IS detected
    - VariantClear() will misinterpret erroneous type
Example V – Direct Type Manipulation

```cpp
inline HRESULT CComVariant::ReadFromStream(IStream* pStream)
{
    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;
}
```
<table>
<thead>
<tr>
<th>Type Name</th>
<th>VARIANT Type</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>VT_EMPTY</td>
<td>Unassigned variable</td>
</tr>
<tr>
<td>NULL</td>
<td>VT_NULL</td>
<td>Null keyword, null keyword</td>
</tr>
<tr>
<td>Integer (Small)</td>
<td>VT_I2</td>
<td>Small literal, CInt(), Asc()</td>
</tr>
<tr>
<td>Integer (Large)</td>
<td>VT_I4</td>
<td>Large Literal, CInt(), Clng(), Literal</td>
</tr>
<tr>
<td>Real Number (Small)</td>
<td>VT_R4</td>
<td>CSng()</td>
</tr>
<tr>
<td>Real Number (Large)</td>
<td>VT_R8</td>
<td>Large Literal, Large Literal, parseFloat()</td>
</tr>
<tr>
<td>Currency</td>
<td>VT_CY</td>
<td>CCur()</td>
</tr>
<tr>
<td>Date</td>
<td>VT_DATE</td>
<td>Date Literal, CDate()</td>
</tr>
<tr>
<td>String</td>
<td>VT_BSTR</td>
<td>Literal String, CStr(), Chr(), Literal String</td>
</tr>
<tr>
<td>Automation Object</td>
<td>VT_DISPATCH</td>
<td>Any Object, Any Object (including arrays, dates, etc)</td>
</tr>
<tr>
<td>Boolean</td>
<td>VT_BOOL</td>
<td>True or False keyword, CBool(), true or false keyword</td>
</tr>
<tr>
<td>Byte</td>
<td>VT_UI1</td>
<td>CByte()</td>
</tr>
<tr>
<td>Array</td>
<td>VT_ARRAY</td>
<td>VT_VARIANT</td>
</tr>
<tr>
<td>Reference to Value</td>
<td>VT_BYREF</td>
<td>VT_VARIANT</td>
</tr>
<tr>
<td>Reference to Array</td>
<td>VT_BYREF</td>
<td>VT_ARRAY</td>
</tr>
</tbody>
</table>

Key: VBScript = blue, Javascript = red
- **Examples of Scripting types**

  **VBScript Examples:**
  ```vbscript
  var = 12
  var = &H12345678
  var = CDate("April 19, 2001")
  var = document.getElementById("pluginTag")
  SomeFunction Array()

  var = "Hi"
  SomeFunction var
  SomeFunction (var)

  Dim var(10)
  ...
  SomeFunction (var)
  ```

  - var is a VT_I2
  - var is a VT_I4
  - var is a VT_DATE
  - var is a VT_DISPATCH
  - parameter is a VT_ARRAY|VT_VARIANT
  - var is a VT_BSTR
  - parameter is a VT_BYREF|VT_VARIANT
  - parameter is a VT_BSTR
  - var is a VT_ARRAY|VT_VARIANT
  - parameter is a VT_BYREF|VT_ARRAY|VT_VARIANT

  **JavaScript Examples:**
  ```javascript
  myVar = 1;
  myVar = true;
  myVar = parseFloat(1.2);
  myVar = new Array();

  ' myVar is a VT_I4
  ' myVar is a VT_BOOL
  ' myVar is a VT_R8
  ' myVar is a VT_DISPATCH
  ```
Plugins that interact with scripting languages / DOM can be used to generate unique objects

- ActiveX objects
- .NET (XBAPs)
- Silverlight
- Flash
- Java

<table>
<thead>
<tr>
<th>Product</th>
<th>Type Name</th>
<th>VARIANT Type</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAPICOM.Utilities</td>
<td>Byte Array</td>
<td>VT_ARRAY</td>
<td>VT_UI1</td>
</tr>
<tr>
<td>ActiveX Control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NET (XBAPs)</td>
<td>COM Objects</td>
<td>VT_UNKNOWN</td>
<td>Any object without [ComVisible (true)] attribute</td>
</tr>
<tr>
<td>.NET (XBAPs)</td>
<td>Arrays</td>
<td>VT_ARRAY</td>
<td>(any base type)</td>
</tr>
<tr>
<td>.NET (XBAPs)</td>
<td>Variable Sized Structures</td>
<td>VT_RECORD</td>
<td></td>
</tr>
</tbody>
</table>
Example: Generating an array of IDispatch objects

- Required for aforementioned DOM vulnerability
- Base scripting languages no help
- Solution: embed a .NET XBAP

```csharp
// LoadCompleted function for a WebBrowser object in the XBAP
void browser_LoadCompleted(object sender, NavigationEventArgs e)
{
    object[] args = new object[1];
    TestClass[] arr = new TestClass[10];
    WebBrowser b = (WebBrowser)sender;

    // Create an array of TestClass classes
    // Note: TestClass must have [ComVisible(true)] attribute to be
    //       packaged as an IDispatch (Variant type 0x0009)
    for (int i = 0; i < 10; i++)
        arr[i] = new TestClass();

    // Array 'arr' will be marshaled as a
    // VARIANT array of IDispatch objects (0x2009)
    args[0] = arr;
    b.InvokeScript("blah", args);
}
```
TRANSITIVE TRUST
• **Browser has an evolutionary security architecture**
  - Core security features
  - Adapted over time to meet changing needs / technologies

• **Components become loopholes for altered security requirements**
  - Previously secure components are now a security threat
  - Weren’t designed with new security features in mind
- Plugins provide additional complications
  - Functionality becomes useful for subverting security mechanisms
  - Plugins can load other objects

- Trust extension becomes transitive in nature
  - Browser explicitly trusts Plugin A, Plugin A trusts Object B
  - Browser inadvertently trusts Object B
Example – killbits and ActiveX

- Only allows instantiation of certain ‘safe’ COM objects
- Many controls are vulnerable just by instantiating them
- Fix: Killbit them

<table>
<thead>
<tr>
<th>GUID</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>47C6C527-6204-4F91-849D-66E234DEE015</td>
<td>Srchui.dll</td>
</tr>
<tr>
<td>35CEC8A3-2BE6-11D2-8773-92E220524153</td>
<td>Stobject.dll</td>
</tr>
<tr>
<td>730F6CDC-2C86-11D2-8773-92E220524153</td>
<td>Stobject.dll</td>
</tr>
<tr>
<td>2C10A98F-D64F-43B4-BED6-DD0E1BF2074C</td>
<td>Vdt70.dll</td>
</tr>
<tr>
<td>6F9F3481-84DD-4B14-B09C-6B4288ECCDE8</td>
<td>Vdt70.dll</td>
</tr>
<tr>
<td>8E26BFC1-AFD6-11CF-BFFC-00AA003CDFDC</td>
<td>Vmhelper.dll</td>
</tr>
<tr>
<td>F0975AFE-5C7F-11D2-8B74-00104B2AFB41</td>
<td>Wbemads.dll</td>
</tr>
</tbody>
</table>

What about persistence?

- Resurrect object properties from untrusted stream if control is SFI
- Properties themselves may be COM objects
- Read a CLSID from the stream, instantiate
- Killbit protection + persistence == NULL
  - Provide persistent stream with killbited CLSID / object embedded
  - ???
  - Profit

- Requirements and Limitations
  - ActiveX control must exist that is SFI
    - MSVidCTL (until lately 😏)
    - Flash
  - Control must have a property of type VT_EMPTY / VT_DISPATCH / VT_UNKNOWN
  - Scriptable methods from IDispatch not reachable generally
  - IPersist* interfaces are reachable

- Custom IPersist*::Load() methods that call CoCreateInstance() also vulnerable
Remediation of ATL Vulnerabilities

- **Things that make the ATL difficult to patch**
  - Most COM objects use the ATL
  - Compiled into the binary, and different each time
  - Many vulnerable code constructs
    - ATL 2.0 (1997)
    - ATL 9.0 (2008)

- **General remediation for 3rd parties**
  - Use PROP_ENTRY_TYPE_* entries in the Property Map
  - Don’t use VT_EMPTY in the Property Map
  - Restrict your control properties to allow a minimal list of CLSIDs
- Example – Out of process plugins and “Protected Mode”
  - Protected mode runs browser at “low integrity” level
  - Many plugins registered to launch at medium integrity level
  - Exposing any functionality from these plugins provides potential privilege escalation

- Java runs out of process
  - Java registers several binaries to run at medium integrity level
  - The jp2launcher.exe program is used to indirectly launch the Java.exe
  - Uses a path supplied from the command line, appends bin\Java.exe
  - CATCH: Not actually as useful as it appears..
  - Reason: downloaded binaries have low integrity SID in their SACL
Fail? Not quite..
- Local Java class files are run with local privileges
- Can read / write files, execute programs, etc..

So, let’s use the legitimate Java.exe..
- Point to a locally downloaded class file
- Alternatively set useful command line property (such as class path)
- Java application runs with local privileges in “Medium Integrity” mode

Result: If Java is installed, IE8 protected mode does nothing..
- How many other plugins might expose dangerous functionality?

Demo
CONCLUSION
• Interoperability has non-negligible security implications
  - Marshalling is hard
  - Controls interacting with each other create new attack capabilities

• Specific data management tasks give rise to unique bug classes
  - Type Confusion
  - Extensions of trust

• Interoperability layers under-treated for security problems to date
  - Not just in browsers!


• Questions?